

AD-A263 001



Computer Science

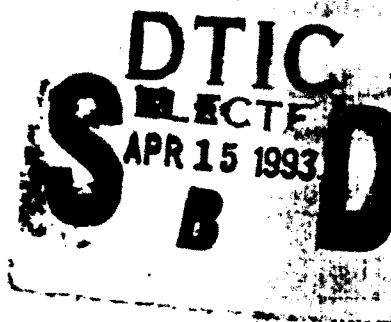
2

Applying High-Level Language Paradigms
to Communications Software for
Distributed Systems

Ellen H. Siegel

January 1993

CMU-CS-93-111



Carnegie
Mellon

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

Applying High-Level Language Paradigms to Communications Software for Distributed Systems

Ellen H. Siegel

January 1993

CMU-CS-93-111

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

CLEARED
FOR OPEN PUBLICATION

APR 1 1993 3

Thesis Committee:
Eric C. Cooper, Chair
Richard F. Rashid
Peter R. Steenkiste
David D. Clark, M.I.T.

DIRECTORATE FOR FREEDOM OF INFORMATION
AND SECURITY REVIEW (OASD-PA)
DEPARTMENT OF DEFENSE

© 1993 Ellen Siegel

This research was supported in part by the Air Force Systems Command and the Defense Advanced Research Projects Agency (DARPA) under Contract F19628-91-C-0128.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

93-07816



131P1

93

3

8

042

~~93~~

~~3~~

~~8~~

~~068~~

93-5-1042

Keywords: distributed systems, communication, high-level programming languages

Applying High-Level Language Paradigms to Communications Software for Distributed Systems

Ellen H. Siegel

January 1993

CMU-CS-93-111

DTIC QUALITY INSPECTED 4

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Thesis Committee:
Eric C. Cooper, Chair
Richard F. Rashid
Peter R. Steenkiste
David D. Clark, M.I.T.

© 1993 Ellen Siegel

This research was supported in part by the Air Force Systems Command and the Defense Advanced Research Projects Agency (DARPA) under Contract F19628-91-C-0128.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution	
Availability Codes	
Dist	Special
A-1	

93 071

93-04588

93 3 8 042

93 3 8 068

Keywords: distributed systems, communication, high-level programming languages



School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

*Applying High-Level Language Paradigms to
Communication Software for Distributed Systems*

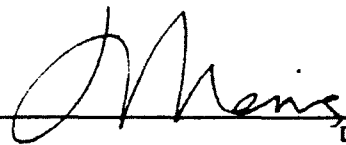
ELLEN SIEGEL

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

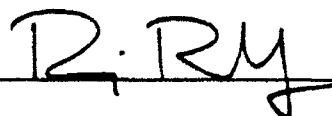

THESIS COMMITTEE CHAIR

1/28/93
DATE


DEPARTMENT HEAD

2/1/93
DATE

APPROVED:


DEAN

2/1/93
DATE

Abstract

Many traditional approaches to systems software do not adapt well to the complexity and scale of distributed systems, but realities of scale and the growing dependency on electronic communication and resource sharing nevertheless make some degree of distribution inescapable. Among other things, distributing systems software introduces complex failure modes and event orderings as well as issues related to heterogeneity and communication latency. Language designers have been investigating techniques for expressing complex systems more cleanly and effectively: various combinations of language mechanisms such as a parameterized module system, first-class functions, abstract types, exceptions, polymorphism, and a strong static type system can also be applied to great advantage in the design, implementation and maintenance of distributed systems.

The dissertation evaluates the benefits of high-level language support in the design and implementation of communications software for distributed systems. Three case studies illustrate the effects of distributed system design and implementation within the framework of a type-safe high-level language with advanced language mechanisms: a remote procedure call system, a distributed Linda system, and a protocol processing framework. The case studies are designed and implemented using Standard ML of New Jersey, which supports many of the relevant language features. Analysis of the design and implementation processes illustrates the benefits and drawbacks of using these advanced language features both individually and in combination to support distributed systems.

Acknowledgments

I would like to gratefully acknowledge the support, both academic and otherwise, that was generously provided by numerous people over the course of my stay at CMU.

I thank my local committee members, Eric Cooper, Rick Rashid, and Peter Steenkiste, for their guidance and feedback over the years. I particularly thank Dave Clark, my external committee member, for his valuable help, feedback and advice far beyond the call of duty.

I am indebted to a number of people for comments and suggestions on various drafts of the thesis and related work. I would especially like to thank Mike Kazar, Greg Morrisett, Scott Nettles, Robert Sansom, Karen Sollins, and Dave Tarditi.

Special thanks are due to Sharon Burks for her amazing skill and apparently untiring efforts in *smoothing over so many of the inevitable bumps of graduate school life*, and to Barbara Lazarus and Indira Nair for the availability of their timely advice and support.

My various office-mates and friends leave me with fond memories of camaraderie and good times which helped to remind me that there could be a life even during graduate school.

Finally, I offer an inadequate but heartfelt thanks to the special friends, family and substitute family who have offered unfailing support, encouragement, and friendship, and who have contributed so much to my life in so many different ways.

Contents

1	Introduction	1
2	Background	5
2.1	Terminology	5
2.1.1	Language Mechanisms	5
2.1.2	Communication	12
2.2	Related Work	14
3	Remote Procedure Call System	18
3.1	Motivation	18
3.2	Remote Procedure Call	20
3.3	Design and Implementation	22
3.3.1	ML-RPC	22
3.3.2	RPC Interface	25
3.3.3	Transport	30
3.3.4	Presentation	31
3.3.5	Client and Server Stub Modules	33
3.4	Discussion	36
3.4.1	Layering	40
3.4.2	Evaluation Strategies	45
3.4.3	Type System	46
3.5	Conclusion	50
4	ML-Linda	51
4.1	Motivation	51
4.2	Linda Overview	52
4.3	Design and Implementation	52
4.3.1	ML-Linda	52

4.3.2	Linda Runtime Types	53
4.3.3	Linda Front-End	56
4.3.4	Tuple-Space	57
4.3.5	ML-Linda Configurations	60
4.3.6	Distributed Tuple-Space	63
4.4	Discussion	67
4.4.1	Layering	67
4.4.2	Evaluation Strategies	67
4.4.3	Type System	68
4.5	Conclusion	68
5	Protocol Processing	71
5.1	Motivation	71
5.2	Overview	72
5.3	Design and Implementation	73
5.3.1	Runtime Types	74
5.3.2	Communication Layer Composition	74
5.3.3	Frames	76
5.3.4	Frame Management	76
5.3.5	Application Client and Server	79
5.4	Discussion	80
5.4.1	Layering	81
5.4.2	Evaluation Strategies	81
5.4.3	Type System	83
5.5	Conclusion	84
6	Performance	85
6.1	SML/NJ	86
6.1.1	SML/NJ Baseline Measurements	87
7	Discussion	92
7.1	Flexibility	94
7.2	Control Flow	96
7.3	Type Issues	97
7.4	Future Work	99
7.5	Contributions	100
7.6	Conclusions	101
	Bibliography	103

A	SML	109
A.1	Syntax and Semantics	109
A.2	Implementation	116
B	Data for SML and C Baselines	117

List of Figures

2.1	OSI Reference Model	13
3.1	Remote Procedure Call Overview	21
3.2	Main Module Dependencies in Client-Side RPC	23
3.3	Main Module Dependencies in Server-Side RPC	24
3.4	RPC Signature	25
3.5	PEER Signature and Peer Functor	27
3.6	Control Flow of Client-Side RPC	28
3.7	Framework of RPC Functor	30
3.8	RPC Functor Instantiation	31
3.9	Transport Signature	31
3.10	Transmissible Types	33
3.11	Presentation Signature	34
3.12	Client Stub Functor Definition	36
3.13	Sample Client Stub Code for store Operation	37
3.14	Server Stub Signature	37
3.15	Simple Server Loop	38
3.16	Server Stub dispatch Routine	39
3.17	Server Stub store Routine	39
3.18	High-level Language Features in ML-RPC	40
3.19	"Wrapper" Functor for Sockets	42
4.1	Local Linda Configuration	53
4.2	Remote Linda Configuration	54
4.3	Distributed Remote Linda Configuration	54
4.4	ML-Linda Types	55
4.5	An ML-Linda Tuple	55
4.6	Tuple-Space Signature	56
4.7	ML-Linda Client eval Operation	58

4.8	ML-Linda Tuple Matching Algorithm	59
4.9	ML-Linda Tuple Matching Algorithm	61
4.10	ML-Linda Local Instantiation	62
4.11	ML-Linda Remote Instantiation	62
4.12	ML-Linda Distributed Instantiation	63
4.13	Multiple Clients Using Distributed Linda Tuple-Space	64
4.14	Signature for Server Distribution Module init and out Operations	66
4.15	Server Distribution Module rd, rdp and rd.done Operations	67
4.16	Server Distribution Module l.in, inp, purge and restore Operations	70
4.17	High-level Language Features in ML-Linda	70
5.1	Protocol Processing Framework Overview	73
5.2	Transmissible Types	75
5.3	Receive A Message	75
5.4	Set Up Message Checksum Closures	75
5.5	Frame Header	76
5.6	Frame Transmission Functions	78
5.7	High-level Language Features in Protocol Framework	80
6.1	Constructing SML/NJ Closures with N Arguments	89
6.2	Allocation Times	90
6.3	Procedure Call Times	91
7.1	Modularity Support	95
7.2	Evaluation Strategy Support	96
A.1	SML Structure Definition	115
A.2	SML Functor Definition	116
B.1	Curried Function Application	117
B.2	SML Allocation Test	117
B.3	C Allocation Test	118
B.4	SML Procedure Call Test Loop for 3 Arguments	119
B.5	C Procedure Call Test Loop for 3 Arguments	119

Chapter 1

Introduction

The dissertation evaluates the benefits of applying high-level programming languages and their associated language mechanisms and constructs to the design and implementation of distributed systems software, focusing especially on communication software. Specifically, it considers the design and implementation of three case studies: a remote procedure call system, a distributed Linda system, and a protocol processing framework. The case studies were chosen to represent a range of distributed communication-related systems software. The wide acceptance of the RPC abstraction makes it a somewhat canonical distributed system component, and provides a familiar and well understood reference point for later discussion. Linda provides a less standard model of communication and data management, but is sufficiently well-known to qualify as a practical application. The network processing framework case study attempts to model some of the strategies being proposed for the design and implementation of next-generation protocols. The design and implementation experience is used to analyze the impact of the high-level language medium on the systems in question, particularly in terms of such criteria as flexibility, extensibility, and performance.

Experience with numerous existing programming languages has demonstrated some of the theoretical and practical utility of versatile language mechanisms and constructs. Language designers have been investigating techniques for expressing complex systems more cleanly and effectively: in fact, many of these constructs can be viewed as finely-tuned applications of the more general abstractions of modularity and encapsulation. Various combinations of language mechanisms such as a flexible, parameterized module system, first-class functions, closures, a sophisticated type system and strong type-checking, abstract types, exceptions, and polymorphism can be used to great advantage in the design, implementation and maintenance of distributed systems.

Parameterized module systems provide flexibility in the form of module-level polymorphism and link-time module reconfiguration. First-class functions and their associated closure mechanism provide a type-safe way of modularizing the control flow of programs by encapsulating the function body with its execution environment. A sophisticated type system provides strong guarantees about

the correctness of type-checked programs; support for abstract and user-defined types extends the protection of the language and type system to a more complex environment tailored to the needs of the system. Type-safe separate compilation can extend the type-safety guarantees of modularity through the compilation and linking phases. Exception mechanisms provide a clean, customizable and type-safe way to propagate error information, even across module boundaries. Various language mechanisms can also be used to specify explicit concurrency.

Each of these language mechanisms increases the design choices of the system implementor, but in many cases the real advantages come from using them in combination. More advanced programming languages provide a unified language framework which can integrate these in a formal, type-safe way. This makes the whole worth more than the sum of the parts, since the formal semantics provided by the advanced language framework guarantees that each particular behavior is defined even when it is used in combination with other mechanisms. This type of guarantee, at another level, is implicit in most distributed systems. By constructing system components whose behavior is formally defined by the language, and which work to extend the language semantics as far as possible across system and machine boundaries, we can achieve both more flexibility and have more faith in our ability to reason about the behavior of increasingly complex distributed systems.

Advanced language constructs can greatly enhance the power and flexibility of distributed systems, where standard systems issues are compounded by reliance on remote communication and the existence of additional degrees of freedom. These manifest themselves, for example, as complex failure modes and independently administered or heterogeneous system components. Distributed systems also require the ability to deal with multiple event orderings and relatively large latencies, thereby increasing the relevance and utility of alternate evaluation strategies such as delayed evaluation.

Advanced programming languages also have some drawbacks, at least in their current state of development. The main drawbacks involve performance and conflicts with the type system in the domain of remote communication. Performance is an important issue in distributed systems in general, and it is an area in which advanced language implementations have often been deficient. However, the efficiency of languages like Scheme and Modula-3 indicate that many such performance drawbacks are implementation specific rather than inherent in the language mechanisms themselves. Furthermore, in many cases there are known techniques for solving many of the specific performance bottlenecks. This is visible, for example, in the ongoing evolution of Standard ML of New Jersey: as more systems-oriented applications are being developed, there is increasing incentive to tackle the important performance bottlenecks. Some of the work in progress includes techniques for improving the efficiency of garbage collection[47], techniques for improving locality of reference in heap accesses, and incorporating domain-specific knowledge into memory management to reduce unnecessary paging behavior[15].

Many of the typing conflicts arise because of the need to communicate beyond the scope of the local environment or address space. With strong static type-checking, the language can make guarantees about the behavior of objects within its own domain, and consequently has no need to

retain runtime type information. Remote communication involves exporting and importing objects to and from foreign environments, and therefore requires sufficient runtime type information to perform message marshaling and unmarshaling as well as sufficient trust or authentication to inject untyped messages into its local environment as trusted, typed objects.

Layering is often a natural model for distributed system design. A certain amount of logical modularity is inevitable in distributed systems, since the physical separation of system components enforces the restriction that they be accessed only via their exported interfaces. Thus, communication is effectively a hidden layer connecting the components of a distributed application. Language supported modularity in the form of parameterized modules makes layering a feasible, type-safe way to support reconfiguration, code re-use, and rapid prototyping. In layered architecture implementations, for example, allowing program modules to reflect the protocol layers greatly increases the clarity of the implementation. With language supported modularity and encapsulation in the form of first-class functions, we can introduce closures into the layer interfaces and provide the basis for function composition and alternate evaluation strategies in a type-safe modular way, providing an opportunity to improve program efficiency by using domain-specific knowledge to control the control flow of the system.

Although the arguments in the thesis apply to the language mechanisms and constructs in the abstract, the case study methodology necessitates the choice of a particular implementation platform. Although a number of languages support various subsets of the relevant language mechanisms, Standard ML[43] (SML or simply ML) and the Standard ML of New Jersey (SML/NJ) compiler[5] were selected. SML's formal semantics and type safety combined with its incremental approach to constructing large programs make it an attractive candidate for building complex distributed or parallel programs. In addition to having the most complete set of the desired language mechanisms, SML has the additional advantages of a body of ongoing research in several important directions. Some of this research includes language extensions, both from a language theoretical and practical usage point of view; performance improvements; and as a vehicle for the implementation of real applications. Despite its many advantages, SML also has some important drawbacks. It does not currently support any form of runtime type information, and its performance and memory usage characteristics are still problematic.

The application of high level languages to systems programming draws together distinct specializations, including distinct sets of terminology and ideology. Chapter 2 provides a brief tutorial on some relevant background and terminology used throughout the body of the thesis, including some specific to SML/NJ, and reviews related work. Chapters 3, 4 and Chapter 5 explore each of the case studies in turn, presenting details of the design and implementation and detailing the associated application of the relevant language constructs¹. Chapter 6 presents some baseline measurements of basic operations in SML/NJ and C, along with a brief discussion of some performance

¹Since examples are presented in SML/NJ syntax, these include some brief ML tutorial segments; more details on ML syntax and semantics can be found in Appendix A.

issues. Chapter 7 provides some global discussion analysis based on the detailed results presented in previous chapters, and concludes with a discussion of contributions and future work.

Chapter 2

Background

This chapter provides some background helpful in understanding the rest of the thesis. The discussion is divided by topics, so readers can skip over topics in their areas of expertise. Section 2.1 defines some of the relevant terminology used in the thesis¹. Some of these terms are used in conflicting ways in different computer science disciplines, so these definitions can also be used to resolve any ambiguity. Section 2.2 describes other work related to systems applications of high-level languages as well as to various sub-topics of the thesis.

Examples are given in SML syntax. High-level descriptions are embedded in the text: paragraphs of SML tutorial in the text are offset and italicized so that they can be easily identified as such. Appendix A provides a more detailed introduction to SML syntax and semantics.

2.1 Terminology

2.1.1 Language Mechanisms

Evaluation Strategies

Parameters There are two basic evaluation techniques applied to parameter and variable binding. A language employs *lazy* or *non-strict* evaluation if it evaluates the value of an expression (and memoizes it, or stores it away for future reference) only when it is needed. The memoization keeps the value from being recomputed on successive references. *Eager* or *strict* evaluation means that an expression is evaluated when it is first encountered.

Lazy evaluation has a side effect related to program termination. If an expression contains a clause which has an undefined value, strict evaluation will result in the value of the entire expression being undefined. With lazy evaluation, however, the undefined clause causes an undefined result

¹ Several of the definitions were taken from Sethi [53], Field and Harrison [22], Watt [61], and Reade [50].

only if it is actually executed; this makes it possible to return defined results even in the presence of undefined clauses. As an example, consider an *if* statement whose conditional expression is a disjunction of two or more clauses:

```
if (n < 0) orelse ((n div x) > 20)
  then ...
  else ...
```

The statement may be legal code, but in the case where *x* is equal to zero the second clause of the condition is undefined² and will cause a division by zero exception. Under strict evaluation, the exception will be raised any time the value of *x* is zero. Under non-strict evaluation, the second clause will not be evaluated in cases where the value of *n* is less than zero, since its evaluation is not required to produce the value of the conditional; in this case, the enclosing function may still return a well-defined value.

Expressions Evaluation strategies can be broken down into three broad categories: lazy evaluation, delayed evaluation, and eager evaluation. There are many specific variations within these categories, but for simplicity we choose to consider them as generic categories. We look beyond their usual application in parameter evaluation, and consider the strategies across a wider domain of distributed system functionality.

Lazy evaluation is an evaluation strategy in which an expression is evaluated and memoized (stored for future reference) at its first use rather than at its invocation. If the value is never accessed, it is never evaluated. Lazy evaluation can enhance performance in cases where the lazily evaluated functions represent significant computation time and/or resources to produce results which may never be required; in such cases, performance is improved by eliminating the processing time for unused data.

Delayed evaluation is similar to lazy evaluation except that it is assumed that the processing will eventually take place. The evaluation takes place in idle cycles or on demand, rather than at the time of invocation, and the results are memoized. It is also useful for increasing parallelism in a program, since a lazily evaluated function can be handed off to another processor to be executed in parallel with the enclosing instruction sequence. The ability to delay execution arbitrarily also makes it possible to recombine operations in such a way as to minimize data accesses [26], which can result in significant performance improvements. The *future* construct in Multilisp [33] is an example of delayed evaluation.

An *eager evaluation* strategy is one in which all functions are executed to completion at invocation, and in fact may result in the generation of even more data than is explicitly required. Eager evaluation strategies are often used in remote data accesses or caching strategies, where the

²*div* is the SML operation for integer division

implementation exploits the assumptions of locality of reference to pre-fetch data in the hopes of avoiding transfer latency for the next request.

Functions Treating a function of n arguments as a successive application of n single-argument functions is called *currying*. A curried function may be used to create a new function object by applying it to a strict subset of the original argument list. This is an application of higher-order functions which is sometimes called *partial application*. For example, consider a curried function f which takes two integer arguments and returns their sum:

```
- fun f (x: int) (y: int) = x + y;  
val f = fn : int -> int -> int
```

The lines preceded by a '-' are the user input to SML, and the lines immediately following are SML's representations of the result types. f is defined as a curried function which, when applied to two consecutive integer arguments, computes their sum. The SML interpreter shows that f is a function (using the keyword `fn` rather than attempting to represent the function body), whose type specification is `int -> int -> int`. The `->` symbol separates the arguments from the return values of a function in a type specification; this particular specification can be read as "a function which takes an integer argument, and returns a function taking an integer argument which returns an integer".

Note that it is not always necessary to specify argument types because of SML's inductive type mechanism, but since `+` is an overloaded function it does not provide enough type information for induction in this case.

The above example defines a function f , which SML recognizes as a curried function taking two successive integer arguments and returning a result of type integer. The expression $\{f\ x\ y = x + y\}$ is equivalent to $\lambda x. \lambda y. (x + y)$ in λ -calculus notation. Applying f to the integer 3 yields a new function, g , which takes a single integer argument and returns the value of its argument incremented by 3.

```
- val g = f 3;  
val g = fn : int -> int  
- g 4;  
val it = 7 : int
```

Continuations are functions representing the "work remaining to be done" in a partially evaluated function application. In general, a continuation may be viewed as a mapping that, when applied to the additional inputs required by the computation, will yield the final result.

Control Constructs

An *exception* is an indication or *signal* that an exceptional condition has arisen. The operation that issues the signal is said to *raise* the exception. Exceptions can not be ignored: if an exception is raised, the program will halt unless a corresponding *handler* has been provided. However, every

exception can be handled, and the programmer has complete control over what the handler does when it is invoked.

An exception facility provides a clean way of propagating error information across arbitrary numbers of program layers. Since exceptions are propagated to the nearest enclosing handler, each program layer has the flexibility to either intercept exceptions or to allow them to propagate through directly.

For example, the following code declares an exception called `MyDiv` and a function `div_by_y`. The function divides its first argument by its second argument. It provides a handler for the system exception `Div`, which is raised by the system on division by zero.

```
exception MyDiv
fun div_by_y (x:int) (y:int) =
    x div y
    handle Div => (print "Division by zero.";
                  raise MyDiv)
```

This code fragment defines an exception `MyDiv` and a function `div_by_y`. `div_by_y` takes two integer arguments, `x` and `y`, and uses the built-in function `div` to compute the (integer) value of dividing `x` by `y`. The keyword `handle` indicates the start of an exception handler. The identifier following it is the name of the identifier being handled (in this case the system-defined exception `Div`, which is raised on division by zero); and the expression following the `=>` symbol is the body of the expression handler. In this case, the handler simply prints an error message and raises its own exception in place of the system-defined one.

If `Div` is raised, the program will print out an error message, and raise its `MyDiv` exception. This construction avoids the overhead of checking the divisor value on every invocation, yet responds to the error condition with a meaningful error message before exiting.

Function composition is a mechanism whereby the results of one function become the input of the next. The symbol ' $\circ$ ' denotes function composition, which is defined mathematically as:

$$(f \circ g)(x) = f(g(x))$$

In other words, the composition of the functions `f` and `g` applied to an argument `x` is equal to the results of applying `f` to the results of `g(x)`. The UNIX *pipe* construct can be considered an example of function composition where all the inputs and outputs are constrained to be characters or sequences of characters. Function composition is an application of higher-order functions.

A function is *recursive* if its body contains a call to itself. Recursion is frequently associated with functional languages, since it is especially useful in the absence of side-effects or assignment. A function is *tail-recursive* if any recursive calls occur only as the last expression to be evaluated in a sequential, eager evaluation. This means that there is no need to store intermediate return addresses, since the called function can return directly to the return address of the current invocation. For example, `fact1` is tail-recursive, while `fact2` is not³:

³For simplicity, these examples leave out tests for negative arguments.

```

fun fact1 x =
  let
    fun f (0, v) = v
      | f (n, v) = f ((n - 1), (v * n))
    in
      f (x, 1)
    end

fun fact2 0 = 1
  | fact2 n = n * (fact2 (n - 1))

```

Although the functions fact1 and fact2 appear to be multiply defined in the above code fragment, the syntax is actually an example of SML's pattern matching syntax (see Appendix A). The possible representations of one or more of the function arguments are enumerated in the cases of the function definition; each case is followed by its version of the function body. The multiple cases are separated by the symbol |. In this case, the pattern-matching is used to recognize when the argument has a value of zero, so that the recursive function can be terminated.

Both implementations are recursive, but fact2 is not tail-recursive because it must perform a multiplication after the completion of the recursive call.

Types and Data

A function is *first-class* if it has the same status as any other value: it can be the value of an expression, or it can be passed as a parameter, returned as a result, or stored in a data structure.

A related concept is that of *higher-order functions*. To be completely general, support for first-class functions should be recursive: a functional argument may itself have a functional argument, and so on. A function is called higher order if either its arguments or its results are themselves functions.

A *closure* is a combination of a function body and its evaluation environment. This composite structure is called a closure because it represents a closed expression, i.e. an expression which contains no free variables. A first-class function may *escape* from its lexical scope by being implicitly or explicitly returned as a function value; any function escaping its scope must be associated with bindings for its free variables.

A function designed to take arguments of mutable type is called *polymorphic*. Polymorphism can be specified via a *type variable*, which can be statically type-checked. The following example defines a function *identity* which takes a polymorphic parameter of type 'a and returns it⁴. The function return type, although unknown, is constrained to be the same as that of its parameter. Invoking the function with an argument of 3 causes 'a to be instantiated as an integer, while

⁴SML uses the single-quote character to tag its type variables; the tagged variables are read as the corresponding Greek letters to distinguish them from regular variables. Thus, 'a is read as *alpha*

invoking it with an argument of `true` instantiates `'a` as a boolean. In both cases, the original parameter becomes the return value.

```
- fun identity x = x
  val identity = fn: 'a -> 'a

- identity 3;
  val it = 3 : int
- identity true;
  val it = true : bool
```

Mutable data is data which can be modified in place. For example, an array whose elements can be individually modified is a mutable array. The structure of the array itself is not modified, but the values of its elements may change. *Immutable* data, on the other hand, will be replaced rather than modified. For example, concatenating a value to the end of a string will create a new string, leaving the original unchanged. A *pure* functional language has no mutable data, since it has no side-effects or assignment statement with which to mutate data.

Memoization is a technique used to avoid recomputation of results. It works by internally storing the results of a specific computation in a table keyed on the expression parameters so that subsequent invocations with the same parameters will result in a lookup rather than a recomputation. For example, a memoized Fibonacci function would take only linear time ($O(n)$) to run because instead of invoking a recursive computation a result would be stored for each intermediate value the first time it was computed.

An *abstract data type* consists of a type definition together with an explicit set of operations, which are the only means by which to compute with values of that type. The type representation and the operation implementations are *encapsulated* within the abstraction boundary; only the interfaces are exported.

Dynamically typed languages [61] only require fixed types for *values*. A variable or parameter may take values of different types at different times. Dynamic typing implies that execution of the program is slowed down by implicit run-time type checks, and also implies that every value must be tagged to identify its type; however, it has the advantage of increased flexibility.

Statically typed languages [61] require that a fixed type is assigned to every variable and parameter at compile time. The compiler uses this information to deduce the type of each expression and to type-check each operation. Statically typed languages are more efficient at runtime because there is no need for run-time type checks, but provide less flexibility than dynamically typed languages. An extension of this is to enable the compiler to deduce types, reducing the burden on the programmer.

Programming Abstractions

Layering is an application of modularity where each module encapsulates an independent logical layer of a program. This approach facilitates configurability by allowing multiple different implementations of each logical layer, with each corresponding to a different program structures and components. It is useful for supporting *transparency*, or the notion that the system functionality is invisible, or *transparent*, from the point of view of the application.

Like many programming methodologies, *modularity* has little to do with the ultimate functionality of a particular program. It does, however, have much to do with other parts of the software life-cycle, particularly the design and the maintenance phases. Modularity has long been recognized as an effective programming technique even for non-distributed systems, and it becomes even more important in distributed systems with independent and potentially mobile components.

Well designed modules also have a tendency to become re-usable program building blocks. Programmers frequently use the same basic data structures to solve many different programming problems. Shared modules have a tendency to be better documented and tuned than private ones, since sharing implies that many people have a need to understand and trust the interfaces and implementation.

Encapsulation is an abstract concept, and as such can be applied to programs written in any programming language. Nevertheless, with the right language support modularity makes a much more effective tool. For example, the definition of the C programming language specifies the behavior of the language constructs, not their implementation. However, because C implementation details are accessible to C programmers and because C does not provide strong type checking, there programmers have been able to rely on implementation dependent features in their programs. This situation creates interactions between the programs and the language modules that are not explicit in the language specification: this makes it difficult to make changes or improvements to the language implementation, and it makes it quite difficult for any programmer not intimately familiar with a program to maintain or debug it.

Programming Methodologies

There are four basic programming methodologies: *imperative*, *functional*, and *object oriented*, and *logic*. This section gives a brief description of the essence of each approach.

The semantics of *imperative* languages are based on the concept of a *store*, or memory abstraction; the programming model for such languages consists mainly of updating the contents of memory [59], or *mutating* program data. This is the programming methodology used in most traditional programming languages, such as Pascal or C.

Pure *functional* programming can be characterized as programming without assignments, or *stateless* programming. Although many functional languages, such as Lisp or SML, are *impure* because they do provide assignment, their programming style is nevertheless dominated by the pure

part of the language and most data is immutable. Another characteristic of functional languages is that users do not have to worry about managing storage for data. Built-in operations on data allocate storage as needed. Storage that becomes inaccessible is automatically deallocated. The absence of explicit code for deallocation makes programs simpler and shorter, but a consequence of this approach is that the language implementation must perform *garbage collection* to reclaim storage that has become inaccessible. Finally, functional programming treats functions as first-class objects.

Object-oriented programming languages, such as SmallTalk, encourage a programming style that relies on the concepts of *inheritance* and *data encapsulation*. The object-oriented programming paradigm is based on the concepts of *object* and *object class*. An object, much like an abstract data type, is a variable *encapsulated* with operations that have the exclusive right to access it. An object class is a set of objects that share the same operations, or *methods*. Inheritance is a language mechanism for defining a new class of objects as an extension of previously defined classes. The new class inherits the public data structures and operations of its ancestor classes. In particular, a *subtype* can be defined by extending or modifying the characteristics inherited from an existing type, or *supertype*. A distinguishing feature of object-oriented programming is captured by the property that an object of a subtype can appear wherever an object of a supertype is expected.

Logic programming languages, of which Prolog is a commonly cited example, was developed primarily for natural language applications and theorem proving. Its name comes from its approach, which is basically an application of mathematical logic rules to a particular set of data.

2.1.2 Communication

OSI Protocol Stack

The standard for structuring hierarchically designed networks is the *Open Systems Interconnection (OSI) Reference Model* [57, 67], based on a proposal from the International Standards Organization (ISO). This model represents network processing as a seven layer hierarchy (see Figure 2.1), where each layer is responsible for a different segment of the processing and has its own view of the data. Logically each layer communicates directly with the corresponding layer on the remote peer, but the actual data and control flow of a transmission proceeds hierarchically from the sending application down the protocol layers to the physical communications medium, and then back up the layers on the remote machine to the receiving application.

Of the seven layers, those of most concern to high level distributed communication software are the *presentation*, *session*, and *transport* layers. The task of the transport layer is to provide reliable host-to-host communication for use by the session layer. It must hide all the details of the communication subnet from the layers above. The session layer is responsible for setting up, managing, and terminating process-to-process connections. It also handles certain aspects of synchronization and recovery. The presentation layer performs various transformations on the data to be sent, such as text compression, byte swapping, conversions to external data representation

format, or possibly encryption.

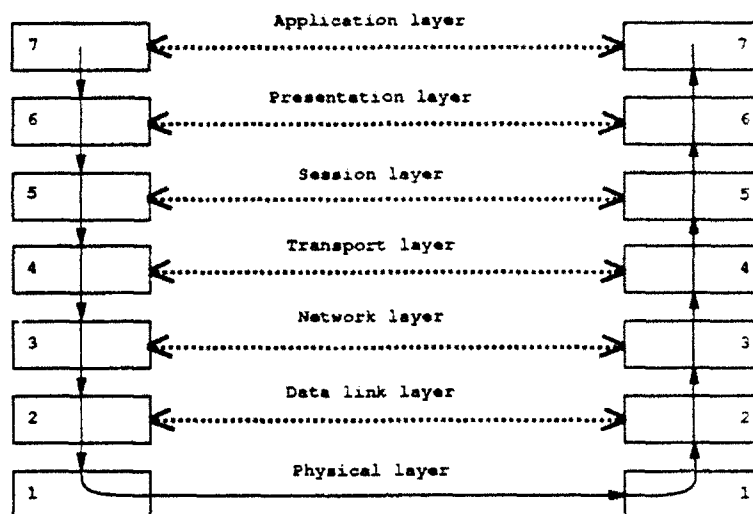


Figure 2.1: OSI Reference Model

The process of linearizing the message data into a form suitable for transmission is referred to as *marshaling*. The specific form of the marshaled data is defined by an external data representation format on which any sender-receiver pair must agree, either statically or via some form of negotiation. The process of restoring the message data to its original form is called *unmarshaling*.

Protocols

Remote procedure call [45] (RPC) is a synchronous communication protocol which extends conventional procedure call semantics to distributed systems: the sending process blocks and waits for a reply from the receiver before continuing. Its familiar semantics make it an intuitive model for programmers, and it has been adopted as the standard communication protocol for many systems⁵.

Message passing usually refers to a high-level asynchronous communication protocol. Messages do not require acknowledgments, so their latency may be as low as a one-way network trip rather than the round-trip time required by RPC. However, many synchronization and reliability issues become the responsibility of the application programmer.

Another useful mechanism for distributed systems is group communication. Remote communication is often directed at groups of remote hosts rather than to a single remote process. Mechanisms

⁵RPC is the standard communication protocol for both SUN's Open Network Computing (ONC) and the Open Software Foundation's Distributed Computing Environment (DCE), which together are run on roughly 85% of UNIX workstations as of 1991.

such as *broadcast*, which sends a message to all connected hosts, or *multicast*, which sends a message to a previously defined group of hosts, can be less complicated, more flexible, and more efficient than an equivalent sequence of sequential calls.

2.2 Related Work

This chapter discusses some systems and protocols that provide groundwork or motivation for various parts of the thesis.

Parallelism

Mach [2] is a shared memory multiprocessor operating system supporting multiple *threads* of control within the same *task*, or address space. It also supports a customizable external pager mechanism which provides the basis for many of the SML garbage collection optimizations.

C-Threads [17] is a user-level thread package implemented for the C [35] programming language on top of Mach. It exports creation and synchronization primitives.

Linda [10, 23] is a programming language veneer intended to facilitate parallel programming. A Linda implementation for a given programming language extends that language with a simple programming model implemented by a handful of new operators which treat communication as movement data in and out of a common, potentially distributed space. The model is conceptually similar to a distributed shared memory implementation, but program and communication data are treated in a uniform way, and are accessed via pattern matching rather than by normal addressing modes. Further details can be found in Chapter 4.

High-Level Languages

Systems Implementation

There has been work on applying advanced language paradigms to deal with heterogeneity in the context of traditional programming languages. The HRPC system [6] enables the interchange of RPC components by fixing a set of component interfaces. The x-kernel [49] is an operating system kernel allowing applications to access existing heterogeneous distributed resources by dynamically choosing the appropriate communication protocol. The Mercury [39] system uses a closure-like mechanism to modify remote port specifications.

Previous work exploring systems software implementation with high-level programming languages includes the Cedar System [58], implemented in Mesa [44] at XEROX PARC; the Topaz system, implemented in Modula-2 [64, 51] at DEC SRC; and the Swift system [13], implemented in CLU [38] at MIT. Various systems software has also been implemented in languages such as Lisp [42], Ada [60], and Argus [40, 41], and is beginning to be implemented in SML [18, 15]. These

systems each provide useful data on the utility of some specific advanced language mechanisms: in particular, the importance of an exception mechanism, explicit concurrency mechanism (e.g. threads), and encapsulation via interface specifications were recurring themes.

All of these languages support some subset of advanced language mechanisms, but each is missing enough to significantly reduce the full benefit of the interactive effects. All but Lisp and SML are imperative, and do not include formal semantics or support for first class functions. Although C++ supports parameterized modules, only CLU, Modula-3, SML and ADA combine them with compile-time type-checking. Although it does not support inheritance or run-time type information, SML has formal semantics and supports a parameterized module system, strong compile-time type-checking, first-class functions, and exceptions: the combination of these language mechanisms combined with its current state of ongoing development make it an attractive platform for this study and for potential future development.

Protocol Architectures

Clark and Tennenhouse propose *Integrated Layer Processing* [14] (ILP) as a structuring mechanism for improving the performance of protocol processing. ILP is a scheme for reducing communication latency in an implementation of a layered protocol architecture by integrating processing components by function rather than by logical layer, collapsing some of the OSI layer boundaries and composing related operations. Some of the modularity of logical layering in the implementation is abandoned in order to more effectively tune the performance.

They illustrate the principle by considering two common protocol operations: checksum in the transport layer, and marshaling operations at the presentation layer. Both operations must touch every byte of data. In a conventionally layered protocol implementation this would require two separate passes over the data, with the data access itself making up a significant fraction of the processing overhead each time. ILP reduces the total cost of the operations by separating the implementation from the layered design, collapsing the layers so that both operations can be performed within the scope of a single data access.

Application Level Framing [14, 20] (ALF) is a strategy of organizing transmitted data into logical packets, or *frames*, meaningful to the application. Conventional packet divisions are arbitrary with respect to application data, requiring communication processing to be strictly sequential. ALF allows greater flexibility and potentially better performance in the control and processing of remote data; in particular, since each packet contains data meaningful to the application, protocol and potentially even application processing can be performed on each arriving packet independently. The work also considers strategies which provide the sender with information about the data representation on the receiver so that final placement information can be incorporated into such *Application Data Unit* packets.

Some effects of delayed protocol evaluation on the performance of a model protocol implemented in C is considered in Gunningberg [26]. The authors compare the performance of a pipelined

invocation of the encoding, checksum and encryption operations to that of the traditional sequential model. Their results show that although their results are dominated by their choice of DES encryption algorithm, the individual results nevertheless indicate improved performance for the pipelined implementation.

Heterogeneity

Heterogeneity poses more problems in distributed systems than just presentation layer processing complexities. Linking previously independent systems together into a loosely-knit distributed system yields exponential combinations of communication protocols, hardware platforms, operating systems, programming languages, and many other features. One attempt to deal effectively with the heterogeneity involves identifying common functionality and designing the system with enough modularity to allow sharing of common code whenever possible [6, 49, 56].

The x-kernel [49] is an operating system kernel dedicated to providing support for multiple communication protocols, all of which export a common application interface. The focus is on using modularity to support diversity, allowing applications to access existing heterogeneous distributed resources by dynamically choosing the appropriate communication protocol.

Morpheus [1] is a language-based approach to protocol implementation based on the x-kernel work, imposing design constraints on the programmer in order to improve the power and efficiency of the programming process and of the resulting protocol implementation.

Communication

There have been numerous RPC implementations. Some of the instances most relevant to the current work deal specifically with heterogeneity and its effect on communication. In particular, the HRPC [6] deals with a heterogeneous machine and system environment, work by Hayes et al. [27] explores issues of mixed-language environments, Gibbons [25] considers a modular, customizable RPC implementation for heterogeneous environments, and Sollins [55] and Herlihy [30] examine issues of external representation formats and translation mechanisms for heterogeneous communication instances.

Work in progress at CMU addresses the issue of support for distributed first-class values in SML [46]. Memory management techniques and lazy transmission are used for efficiency, and standard SML sharing semantics are preserved.

First-class functions have also been used for other aspects of distributed communication. The Mercury [39] system uses a closure-like mechanism to modify remote port specifications. A *port* in Mercury is essentially a specification for a remote function, including parameter, return, and exception types and a location. An interface is composed of a set of ports which specify the functions imported and exported by a service. A client may export a modified port by pre-binding some of its parameters, providing an additional level of interface flexibility.

Fault Tolerance

One of the most widely used fault tolerance mechanisms is *replication*. Replication involves maintaining logically consistent copies of whatever hardware or software system component is being replicated. One form of hardware replication is mirrored disks, where all disk writes are automatically made to two or more identical disks; should the primary disk fail, the system can transparently switch to one of the mirrored copies.

Similar strategies are used for software replication. To maintain strict copy consistency, an operation cannot be permitted to complete until all the relevant replicas have been updated. Much of the challenge in designing replication algorithms is to maintain consistency without trading off too much performance in the process. One approach is to use *quorums* of replicas for accessing and updating objects [29]. To use quorums, all replicas are marked with some form of version number so that it is always possible to identify the most recent copy among any group of replicas. Imagine a data file that is often referenced, but updated only once a day. Suppose we replicate it on three machines to ensure its availability. If we require that all updates succeed atomically on all three replicas, then we ensure copy consistency and it is only necessary to read from one replica. Or, if we require atomic updates to two replicas, then reading from two replicas will ensure that at least one of them will have the most recent version of the data. In general, read and write quorums can be chosen to optimize frequent operations at the cost of less frequent operations; the invariant is that they must always overlap.

Chapter 3

Remote Procedure Call System

3.1 Motivation

Communication is a fundamental part of any distributed system, but it is also the source of many complexities. Many of the issues that make distributed systems difficult to program and maintain are introduced either in the process of communicating from one site to another, or in recognizing and reacting to the wide variety of distributed failure modes. An important role of the systems programmer is to encapsulate the complexities of the systems issues into modular packages that provide services required by applications programmers: for distributed systems, this includes distributed communication and failure modes as well. Ideally, the applications programmer should need a minimum of systems knowledge to effectively integrate the systems packages into a particular application, and the same should be true in the presence of distribution.

In practice, however, much of the complexity of distributed system state escapes from the systems packages through to the application and even all the way back to the user. In conventional programming languages, managing complex state often implies sharing global variables; this makes the application programmer's choice of variable names sensitive to the naming schemes of any imported systems packages. Most conventional languages do not enforce interface specifications, nor do they allow private (nested) procedure definitions or offer protection from implicit dependencies on values of global variables. The interdependencies can be even more complex in a distributed system, where an application programmer may not even have access to all the necessary information about system state: in that case, it is imperative that a systems package be able to reliably encapsulate its private data, so that the integrity both of the system and of the application can be properly maintained. Many of these pitfalls can be avoided with the help of a reliable type system and a built-in module system with language-enforced interface specifications.

The many variables involved in distributed communication in a heterogeneous environment have led most systems programmers to customize implementations in order to optimize for the needs of

their particular systems. Choices such as programming language, machine type, and external representation format all affect the processing required by the communications system; application-specific features such as the amount, complexity, and frequency of data being communicated can have significant performance implications as well. With adequate language support, however, customization need not always imply re-implementation. A system can be a set of tailored building blocks rather than a monolithic structure: an application programmer can customize a system by choosing an appropriate combination of blocks and possibly providing a small amount of "glue" in the right places, using the blocks in much the same way as a C [35] programmer might include UNIX [37] system libraries.

The main difference, from the point of view of the application programmer, is the extra safety and simplification provided by the high-level language support. This type of modular system must be easily configurable and extensible so that it can accommodate a wide range of applications' needs. Language supported encapsulation, both at the language and module level, provides the implementation independence required to support effective modularity: for example, the application programmer as well as any systems module implementors can choose names freely within their own scopes without having to worry about external name conflicts. A parameterized module system provides type-safe module reconfiguration at link time, and the first-class status of functions even allows a degree of type-safe runtime configuration based on records of typed functions¹. A powerful, integrated type system ensures that the imported system modules can only be combined in meaningful ways, and imposes a structure on their associated datatypes which is enforced by the language. The strong static type-checking guarantees local runtime correctness, and careful system design allows some guarantees to be extended even across system boundaries. This ensures that there is virtually no danger of errors due to syntactic misconceptions. Thus, high-level language support can convert a haphazard collection of libraries into a structured system by specifying and enforcing a small set of fixed interfaces, ensuring the correspondence between the exported interfaces and their implementations.

A common conceptual model of a distributed system is that of a non-distributed application with a communication package layered in between the peers, and it is often advantageous to have the implementation take the same approach. The RPC case study makes extensive use of layering abstractions: the flexibility and conceptual simplicity of these high-level language abstractions increases the expressive power and efficiency of the programmer. The main goal is flexibility, in several senses: reconfiguration, extensibility, module re-use, and ease of use (transparency to applications). Configurability is based primarily on a parameterized module system. The parameterized modules provide a kind of higher-level polymorphism, allowing the module implementations to vary but keeping the interfaces fixed. This approach allows different protocols to be represented as multiple implementations of the same signature². Because the module system

¹These function records are similar to the untyped *transfer vectors* used in, for example, I/O routines.

²For example, UNIX sockets and Mach IPC could both be used to implement a transport signature

is parameterized, the available set of modules can be easily recombined at link time to provide the desired combination of protocols and features. Adding a new module implementation need only involve those applications actually using it.

The naming hierarchy imposed by parameterized modules simplifies the re-use of existing code, which implementors often want or need to use in their systems. In principle, of course, module re-use is not restricted to high-level languages. In practice, however, complications in conventional languages with adjusting interfaces, coordinating variable names, and deciphering type representations tend to prevent this sort of sharing except occasionally within fairly tightly-knit organizational groupings.

A strong static type-checking mechanism provides a well-defined, structured programming environment and can greatly simplify the task of the programmer. It also provides strong semantic guarantees about even the runtime type safety of the code. In addition to the usual careless mistakes and typographical errors, complex semantic or conceptual errors often manifest themselves as type errors. Furthermore, language supported encapsulation increases the programmer's control over the local state, and therefore over the ultimate behavior of the program.

We explore the process of designing and implementing an RPC system within the environment of an advanced programming language, considering the advantages and drawbacks introduced by the availability of its powerful language mechanisms. Section 3.3 presents some relevant aspects of the design and implementation of the main modules in the RPC package, and provides some examples of configurations. Section 3.4 discusses the reasons for the interface and implementation decisions, and analyzes the resulting advantages and disadvantages.

3.2 Remote Procedure Call

The RPC [7] model of distributed computation, illustrated in Figure 3.1, is one of communicating clients and servers. A server exports a well defined interface, which may include types, functions, exceptions, and data, and clients access the server's data or services via this exported interface. Servers may in turn be clients of other servers. A remote procedure call package is responsible for handling communication-related tasks as transparently as possible³.

RPC is so named because it is effectively a remote extension of the semantics of the familiar local procedure call interface. Although there are necessary complications such as the more complex error modes introduced by the independent computations, the idea behind remote procedure call is to provide the programmer with an interface to remote communication which is as simple and transparent as possible.

³It is usually desirable to provide the client with some sort of handle associated with the remote peer of any particular connection so that it is capable of interacting with multiple peers concurrently.

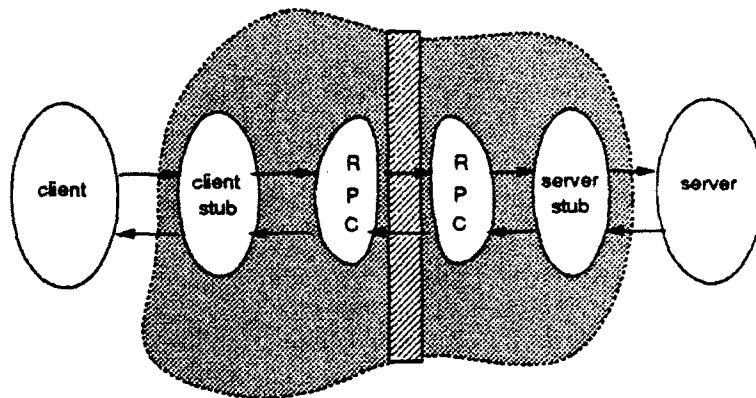


Figure 3.1: Remote Procedure Call Overview

3.3 Design and Implementation

This section presents a description of the SML *signatures* describing the interfaces of the main modules in the RPC system. There are a variety of communication protocols to choose from, but the wide acceptance of the RPC abstraction makes it a somewhat canonical distributed system component, and provides a familiar and well understood reference point for later discussion.

The RPC paradigm maps naturally into SML: the remoteness of the server forces encapsulation of the server internals, constraining all access to come through the exported interface (which is defined by an SML *signature*).

Although the roles of the client and server are distinct, the client and server RPC configurations are roughly symmetric. As illustrated in Figure 3.1, both the client and the server access their local RPC package via a *stub* module. The stub modules are tailored to a specific server interface, and are generally responsible for translating the procedure call types and semantics to and from those appropriate for remote communication. Stub modules are ordinarily automatically generated by a trusted *stub generator*.

Each of the signatures is implemented by one or more SML *functors*, or parameterized *structures*. Some functors export the same signature because of the layered design of the system; functors exporting the same signature look the same from the point of view of any calling modules, and are syntactically interchangeable. A functor definition is instantiated to a runtime structure at link-time by invoking it with instantiated structure parameters, in much the same way as a procedure would be invoked at run-time with parameters of the appropriate type.

The interface and code segments presented in the text are written in SML. The discussion will provide some high level explanation of the examples; SML tutorial segments appear offset in smaller, italicized type. A more detailed tutorial on SML syntax and semantics can be found in Appendix A.

3.3.1 ML-RPC

The core of the RPC package is composed of the RPC, PRESENTATION, and TRANSPORT signatures and a functor implementing the RPC signature. The system also includes signatures for the supporting type abstractions BUFFER and PEER. The package exports basic RPC functionality: client routines for connecting to a remote peer and making a remote procedure call, and server routines for creating and destroying service ports, receiving a request and sending a response. At least one implementation is provided for every signature, although most are intended to be only the basis for a more extensive library of implementations.

The interdependencies among the main modules are depicted in Figures 3.2 and 3.3. The rectangular boxes represent interfaces, and the ovals represent implementations. The shaded modules embody type information, and are not part of the control flow. The sharing of the type modules constrains the structure of the interface types without fixing their representation. Note also that the client stub implementation exports the server interface to the client (or at least the modified version

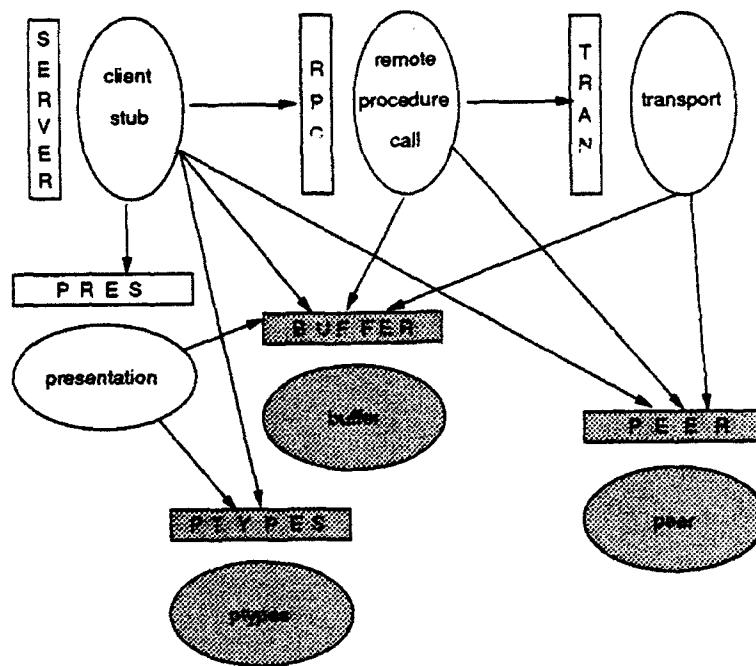


Figure 3.2: Main Module Dependencies in Client-Side RPC

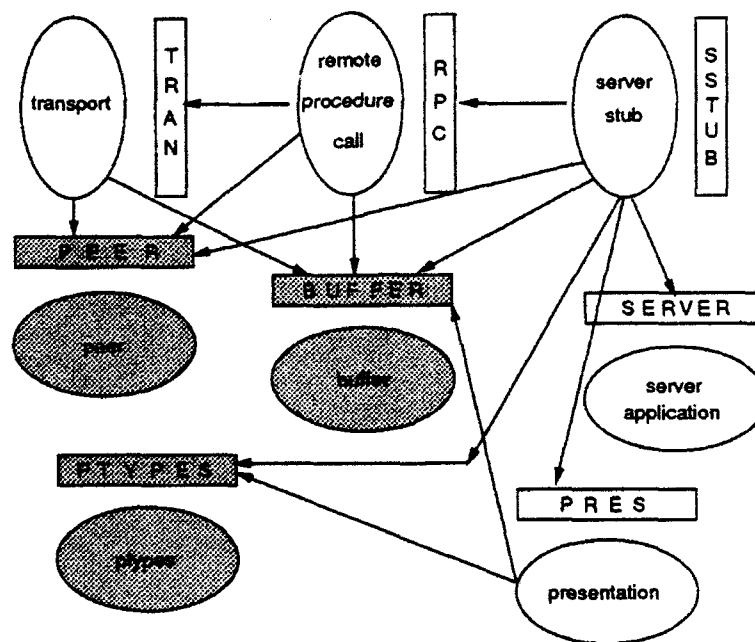


Figure 3.3: Main Module Dependencies in Server-Side RPC

of it produced by the stub compiler).

3.3.2 RPC Interface

Although the exported functionality is typical for an RPC package, the interface itself is not: it makes use of abstract types, first-class function parameters, and polymorphism. The RPC signature is shown in Figure 3.4.

```
signature RPC =
sig
  type rpc_port
  structure Buffer: BUFFER
  datatype msg = MSG of Buffer.buf |
    FN of unit -> Buffer.buf
  exception Bind
  structure Peer: PEER

  (* client-specific routines *)
  val connect : Peer.dest list -> rpc_port
  val rpc : rpc_port -> msg -> msg

  (* server-specific routines *)
  val create : Peer.dest list -> rpc_port
  val recv_req : rpc_port -> rpc_port * msg
  val send_resp : rpc_port -> msg -> unit

  val destroy : rpc_port -> unit
end
```

Figure 3.4: RPC Signature

The RPC signature specifies a conventional RPC interface. In keeping with convention, the signature name is in all capital letters. It first declares its exported types and inherited structures, and then declares its exported operations. The signature exports two types, one abstract (rpc_port) and one explicit (msg); two nested structures, with signatures BUFFER and PEER; and one exception (Bind). The BUFFER and PEER structures provide a way to constrain the representations of the transmitted message and the remote peer, respectively, to be shared among the system modules without fixing them absolutely.

The RPC signature exports a structure Buffer, of type BUFFER. This module is used to represent the transmissible form of the remote message, and all the RPC modules that manipulate

the message explicitly use the `BUFFER` signature in order to constrain the type interfaces to be the same.

The signature also exports an abstract type `rpc-port` representing an RPC connection to a remote peer. It also exports a type, `msg`, with explicitly defined constructors `MSG` and `FN`. The type `msg` is constructed from the abstract type `buf` exported by the `BUFFER` signature, allowing for different concrete representations but constraining that the representation be shared among the relevant modules of the RPC system. By using the appropriate constructor, a message can be specified either explicitly, or as a first-class function which will yield the message when invoked. This provides the caller with an additional degree of control over the evaluation order of the message processing: the message can be constructed before making the call, or the construction can be left to the discretion of the RPC implementation. Delaying the evaluation is safe because the function automatically includes a closure containing bindings for any free variables that are bound within its lexical scope. The signature also exports an exception, `Bind`, which may be optionally handled by any callers.

The RPC signature also exports the `Peer` structure, which is used to represent information about the peer with which the caller is communicating. It specifies types which need to be shared among several different modules, so it is best implemented as a separate module which can be imported as a functor parameter by any modules requiring the definitions. The signature for the `Peer` structure appears in Figure 3.5.

The `PEER` signature exports three types which can be used to identify a remote peer. A host can consist of a string-valued `H-NAME`, an int-valued `H-ID`, or the null-valued constructor `H-DEFAULT`. A port can consist of one of a similar set of port constructors. A `dest` is a record with the fields `rpcpeer`, which takes a value of type `host`, and `rpcport`, which takes a value of type `port`. This set of types can be accessed through the `Peer` structure in the RPC signature. The `Peer` functor definition indicates that the `Peer` functor takes no parameters, and that it conforms to the `PEER` signature.

The function `Rpc.connect` takes a parameter of type `Peer.dest list`, and returns an object of type `rpc-port` representing the new connection. It is used by the client to set up the necessary connection state for communication with a remote peer of which it wishes to make requests; the information necessary to specify the remote peer is encoded in the `Peer.dest list` parameter.

The function `Rpc.rpc` is used by a client to make a request of a server to which it has previously connected. It provides standard RPC semantics, blocking the client until a reply comes back from the server⁴.

Rpc.rpc is a curried function, so applying it to its first argument, the `rpc-port`, will yield a new function. This new function takes a message object of type `msg`, and returns another object of type `msg` representing

⁴Note that in practice the call can be made asynchronous by forking a new thread to make each synchronous RPC call, freeing the client to continue with other computation.

```

signature PEER =
  sig
    datatype host =
      H_NAME of string | H_ID of int | H_DEFAULT
    datatype port =
      P_NAME of string | P_ID of int | P_DEFAULT
    datatype dest = D of {rpeer: host, rport: port}
  end

functor Peer () : PEER =
  struct
    <host, port and dest definitions>
  end

```

Figure 3.5: PEER Signature and Peer Functor

The set of types exported by the PEER signature are not abstract, since they will be used by various modules to construct representations of the desired message destination. The types `host` and `port` are defined with the keyword `datatype` because they are complex types composed of a set of constructors. The constructors, when applied to the (possibly empty) set of associated types, produce a result of the specified type. For example, any of `H_NAME "host1"`, `H_ID 504`, or `H_DEFAULT` would evaluate to an object of type `host`.

A functor is a parameterized structure. Its "type" is a signature identifier, and its parameters are SML structures or functors. A functor must provide an implementation for everything specified in its signature. It may contain additional functionality or definitions, but they will only be accessible from within the functor (or structure) body. Since it contains only public type definitions, the `Peer` signature and structure definitions (i.e., the interface and implementation) will be identical except for the keywords (e.g. `signature` will be replaced by `structure` or `functor`).

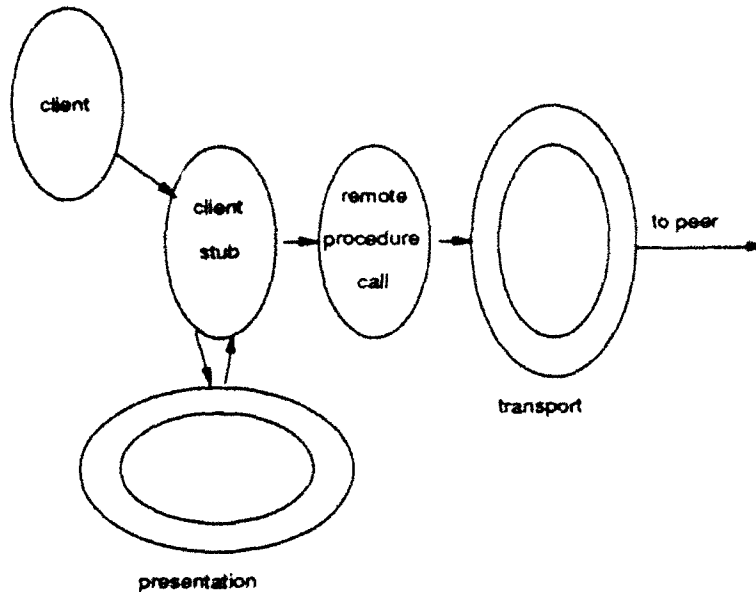


Figure 3.6: Control Flow of Client-Side RPC

the RPC reply. The two functions may be composed: when a curried function is applied to its full set of arguments, the compiler can generally optimize the application into the more efficient tupled argument invocation. However, if the function is applied to only one of its arguments, the result will be a closure which can be bound to a local name and used like any other procedure. If, for example, a program invokes `rpc` several times to the same server, it is possible to save computation by applying the function once to the relevant connection argument and making the subsequent remote calls using the resulting closure with the connection parameter already bound.

```

val reply0 = Rpc.rpc con1 (FN marshal_function)

val host1_rpc = Rpc.rpc con1
val reply1 = host1_rpc (FN marshal_function)
val msg = marshal_function ()
val reply2 = host1_rpc (MSG msg)

```

For example, if `con1` is a `rpc port` object for `host1` and `FN marshal_function` is a message-generating function, a client can invoke an RPC call by supplying the two parameters to the signature function `Rpc.rpc`, where `Rpc.rpc` refers to the value (or function) `rpc` located within the structure `Rpc`. Or, a client could define a function `host1_rpc` by applying `Rpc.rpc` to `con1`, and then proceed to use that function for communication over that particular connection; when applied to an object of type `msg`, `host1_rpc` sends the message via `con1`, and returns the reply.

The function `create` is used by the server to create a request port on which to listen for client request. `create` takes a parameter of type `Peer.dest list` specifying any necessary

information about the server request port, and returns a `rpc_port` object on which the server can listen for requests.

The function `recv_req` is invoked by the server to await client requests; its argument is the server port on which to listen, and it returns a 2-tuple containing the port on which to reply and the request message. `send_resp` is a curried function; it takes the reply port supplied by `recv_req` and a representation of the reply message, and returns `unit`.

The function `destroy` can be used by either the client or the server: it cleans up the state for either a client connection port or a server request port. It takes a parameter of type `rpc_port` and returns `unit`.

The RPC implementation is constrained only in that it must conform to the exported signature. The package also relies on some supporting systems including a presentation module for marshaling basic types, an underlying transport module, and client and server stub modules. The client and server stubs are implemented by parameterized modules, or *functors*, which are generated from the information in the (unmodified) server signature. For our purposes, the stubs play both a marshaling and control role. For each operation in the signature, the client functor contains a routine that constructs (and optionally executes) a function which marshals the parameter values into an external representation, invokes a transport routine to make the remote call, receives the response, and constructs (and optionally executes) a function which unmarshals the reply value(s). The server functor provides a listener loop which repeatedly accepts incoming requests and forks a thread to invoke the proper server operation and transmit the marshaled response back to the client.

Consider an implementation of the RPC signature from Figure 3.4 as a functor with two parameters, a transport module with signature `TRANSPORT`, and a Peer module with signature `PEER` (see Figure 3.5).

The RPC functor is parameterized by a transport structure and two modules defining sets of types: one providing a representation of the remote peer, and one of a buffer abstraction. Client and server stub functors are responsible for translating application data into the proper format for transmission, and they make use of supporting structures such as an implementation of the `PRESENTATION` signature, implementations of the `PEER` and `BUFFER` signatures, and optional modules implementing functionality such as encryption or authentication.

Figure 3.7 shows a possible implementation of the RPC signature. The abstract type `rpc_port` is defined with a datatype which includes types from the `Tran` structure (see Figure 3.9), one of its parameters, and from the local datatype `port_status` (which is not visible outside the functor body) since it does not appear in the signature). The structure `Peer` is bound to the parameter `RPeer`.

To configure the `Rpc` functor to use a socket package implemented by a SML structure (or functor) called `Sockets`, the functor would be instantiated as show in Figure 3.8 to produce a structure named `Rpc_S` (assuming that the structures `Sockets` and `MyPeer` had already been defined). A functor can be invoked an arbitrary number of times; each invocation will yield a distinct structure object. For example, the `Rpc` functor could be used to instantiate a second structure which

```

functor Rpc (structure RPeer : PEER
             structure Buf : BUFFER
             structure Tran : TRANSPORT) : RPC =
struct
  datatype port_status = LIVE | DEAD
  datatype rpc_port =
    C of (Tran.msginfo * port_status ref)
  structure Buffer = Buf
  datatype msg = MSG of Buffer.buf |
    FN of unit -> Buffer.buf
  exception Bind
  structure Peer = RPeer
    < functor body >
end

```

Figure 3.7: Framework of RPC Functor

takes a Mach IPC transport parameter.

3.3.3 Transport

The RPC system defines a basic transport signature, which is one of the parameters of the RPC functor. Its signature is shown in Figure 3.9. Using a standardized interface allows the system to switch transparently among transport implementations. Since it is reasonable to assume that a transport implementation will provide some sort of *send/receive* pair, it is possible to impose this particular interface on a wide variety of transport protocols. The specification of the destination is generalized using the PEER signature described in Figure 3.5; this allows for differences in representation among different protocols and protocol implementations. For the rare transport protocols unable to construct the necessary operations, it is possible to treat unsupported operations specially using the SML exception mechanism⁵.

The TRANSPORT signature exports the type *msginfo*, which is an abstract type with a representation private to each implementation. It is used to hold whatever relevant information the implementation requires to facilitate the message transmission: some possibilities are sender or receiver address representations, or connection state (in a connection-oriented protocol). The message

⁵Unfortunately, this approach could potentially confuse an unwary user by raising a runtime exception for what is technically a type-safe, supported operation.

```

- structure Rpc_S = Rpc ( structure RPeer = MyPeer
                          structure Buf = Buffer
                          structure Tran = Sockets);

- structure Rpc_M = Rpc ( structure RPeer = Peer ()
                          structure Buf = Buffer
                          structure Tran = MachIPC);

```

Figure 3.8: RPC Functor Instantiation

The functor parameters are structures which have been instantiated prior to binding; this can be done independently of the functor instantiation, as with the structure `MyPeer`, or within the functor invocation statement, as with the instantiation of the `Peer` functor within the `Rpc_M` definition. Since `MyPeer` exists independently of the `Rpc_S` structure, its exported types and values can also be accessed independently, whereas the `Peer` object within `Rpc_M` is strictly local to that structure.

itself is represented by a buffer abstraction type `buf`, which is exported from the `Buffer` module. The `TRANSPORT`, `PRESENTATION`, and `RPC` signatures all specify the same `Buffer.buf` type as the message representation, although it is still possible to incorporate arbitrary (existing) transport and/or presentation packages using other message representations by using “wrapper” functors to convert the type representation to the specified format. This level of type checking will be performed at functor instantiation time.

The signature exports two exceptions, `Msg` and `TimeOut`. `Msg` is raised if there is some problem with the message itself, and `TimeOut` is raised if there is a problem with the connection path to the remote peer. The transport signature also exports the same `Peer` structure as the `RPC` signature.

The `create` routine converts a destination specification into the representation used by the particular transport implementation, and returns a `msginfo` object encapsulating the relevant information. Specifying the argument to `create` as a `dest list` provides an additional degree of flexibility: in particular, it allows the specification of entire destination groups for multicast or broadcast communication.

3.3.4 Presentation

The function of the presentation module is to translate data from its local representation to a representation suitable for communication. This almost always involves linearizing data by replacing references with the referenced data, and in a heterogeneous environment often also requires byte-swapping operations to translate between different machine representations. Other operations are

```

signature TRANSPORT =
  sig
    type msginfo
    exception Msg
    exception TimeOut
    structure Buffer: BUFFER
    structure Peer: PEER

    (* state management functions *)
    val create: Peer.dest list -> msginfo
    val connect: Peer.dest list -> msginfo
    val bind: Peer.dest list * msginfo -> unit
    val destroy: msginfo -> unit

    (* transmission functions *)
    val rpc: msginfo -> Buffer.buf -> Buffer.buf
    (* (destination, local_port) -> msg -> unit *)
    val send: Peer.dest list * msginfo -> Buffer.buf -> unit
    (* may signal TimeOut *)
    val recv_b: msginfo -> msginfo * Buffer.buf
    val recv_nb: msginfo -> msginfo * Buffer.buf
  end

```

Figure 3.9: Transport Signature

protocol and implementation dependent, as is the final form of the processed data. Some protocols always convert data to and from a specified *external representation language*, while others may use negotiation to determine the appropriate format for a particular transmission or communication session.

The SML/NJ compiler does not have a mechanism for maintaining runtime type information at the current time, although there are plans to include support for a *dynamic* type in a future release. To temporarily get around the runtime type information problem, type information is encoded by tagging data with the appropriate constructor of an exported datatype definition. The set of available specifiers will not support the full range of possible types, but it is flexible enough for the purposes of the case studies. The type definition is available as an independent SML structure (see Figure 3.10), which can be imported as a functor parameter by any functor requiring the type definitions. In addition to the enumerated type, it exports an exception `Type` which can be raised when any type-related errors are encountered.

```
signature PTYPES =
sig
  exception Type
  datatype T = UNIT | INT of int | STRING of string
              | BOOL of bool | REAL of real | LIST of T list
              | ARRAY of T array
end
```

Figure 3.10: Transmissible Types

The basic operations of the presentation module are `marshal` and `unmarshal`. `marshal` translates language types, enumerated by the exported type `T`, into values of type `xrep` suitable for combining into an outgoing message; `unmarshal` translates `xrep` values back into values of type `T`.

The function `open_msg` initializes the state for unmarshaling an incoming message: it takes the message as a parameter, and returns a value of type `xrep` that is used in subsequent calls to `unmarshal`. `end_message` takes a list of marshaled representations of type `xrep` and composes them into a single object of type `Buffer.buf`, ready to transmit via the transport protocol.

3.3.5 Client and Server Stub Modules

The translation of the application's local call into a remote communication occurs in the *stub* modules. The stub modules provide the interface between the application and the RPC system, and are responsible for invoking the appropriate presentation and transport operations for each application

```

signature PRESENTATION =
  sig
    type T
    structure Buffer : BUFFER
    type xrep
    exception UnknownType
    exception EndOfMessage

    val open_msg: Buffer.buf -> xrep
    val end_msg: xrep list -> Buffer.buf

    val marshal: T -> xrep
    val unmarshal: xrep -> T
  end

```

Figure 3.11: Presentation Signature

request: they apply any necessary transformations to the data (parameters), invoke the remote call by sending the appropriate message to the server, and decode any reply values before returning them to the application. In a production system, these stub modules would be automatically generated from the application interface specification and perhaps some user-specified protocol information. Stub generators avoid the tedious process of regenerating the encoding and decoding routines for each new application routine, and help to avoid programming errors in the low-level data manipulations required for remote communication.

To demonstrate the structure of the client and server stub modules, we consider a simple database application with the signature shown below:

```

signature DB =
  sig
    type vtype
    exception NotFound
    val store: string * vtype -> unit
    val lookup: string -> vtype
    val delete: string -> unit
  end

```

The database server exports an abstract type, `vtype`, the exception `NotFound`, and the three operations `store`, `lookup`, and `delete`. Most of the protocol-specific information, such as

the composition of the application header (the application-level information passed to the server which specifies the client request), is encoded in the stub routines. If a stub compiler is used, this information could be provided as input dynamically, or selected from a set of known definitions. If we configure a system using the RPC package, a stub generator would need to produce four modules: an augmented server signature to be exported by the client stub, a functor implementing the client stub, a server listen-loop signature, and the server stub implementing the listen-loop and the relevant server stub routines.

The client stub implementation would export the augmented server signature, which is generated by simply prepending a connection argument to each server routine:

```
signature DB_CL_STUB =
  sig
    type con
    type vtype
    exception NotFound
    val store: con -> string * vtype -> unit
    val lookup: con -> string -> vtype
    val delete: con -> string -> unit
  end
```

As we can see in Figure 3.2, the client stub module depends on a number of other modules. The stub implementation can access these supporting modules as functor parameters, and use them in the type, variable, and procedure definitions in its implementation. Figure 3.12 shows a possible partial implementation of a client stub module for the database server.

The client stub code for a call to *store* would follow a basic pattern: it would marshal the message header and any parameters, compose the result into a message of the appropriate type, send the message, and wait for and unmarshal a reply. The marshaling and unmarshaling operations require calls to a presentation module, which is one of the stub module's parameters. The sending and receiving involve calls to the RPC module, a second parameter. For a simple application header consisting of string representing the operation name and an integer representing a uniquifier (generated by the function `next_unique`), the client stub code for the *store* operation might look as follows:

The stub operation above is simplified for clarity. The implementation details are not constrained by the signature. For example, the stub might have additional parameters which provide various authentication or encryption functionality, or other functions relevant to a particular application.

On the server side, the top-level module is the server stub functor which contains the server listen-loop. The functor implementation contains a dispatch routine and marshaling routines for each operation exported by the server signature, but only the server listen-loop is exported by the signature. While the client application is parameterized by the stub, the opposite is true on the server

```

functor DbClientStub (structure Pres : PRESENTATION
                     structure Buf: BUFFER
                     structure Peer: PEER
                     structure Rpc : RPC
                     structure Types : PTYPES)

: DB_CL_STUB =
struct
  type con = Rpc.rpc_port
  type vtype = Types.T
  exception NotFound
  open Types

  < stub body >
end

```

Figure 3.12: Client Stub Functor Definition

The type con is defined as an Rpc.rpc_port, and the type vtype is defined as a Types.T. The keyword open is used to make the identifiers from the signature of the named structure available without the need to use the '.' notation: in this case, it makes it simpler to use the type constructors in the body of the functor.

side: the server stub functor is parameterized by the server application module, since the application routines are invoked as a result of receiving a client request.

The store routine in the server stub is basically a reverse implementation of that in the client stub. It unmarshals its arguments, invokes the appropriate application operation, and returns the marshaled result, if any.

The listen-loop will then pass the marshaled response back to the RPC package, and it will make its way back through the system layers until it is returned to the client application routine in the appropriate form.

3.4 Discussion

As might be expected, the ML-RPC package provides roughly the same functionality as that of any other RPC package. Using the language mechanisms of SML, however, ML-RPC is able to achieve much of its functionality in a more flexible and consistent way by taking advantage of the language mechanisms provided by a high-level language.

```

fun store con (key, data) =
  (Rpc.rpc con
    (MSG (Pres.end_msg
      [Pres.marshal (STRING "store"),
        Pres.marshal (INT next_unique()),
        Pres.marshal (STRING "key"),
        Pres.marshal data])));
  ())

```

Figure 3.13: Sample Client Stub Code for store Operation

The `store` operation is a curried function which takes a connection and a tuple consisting of a key and a data object. One possible implementation marshals the message components in place and passes the marshaled message to the `Rpc` structure. The application header in this case is composed of a string operation name (used as the dispatch argument by the server stub) and an integer uniquifier. The stub marshals the application header and each of the arguments, composing then composes them into a list which it passes to `Pres.end_msg` after applying the constructor `MSG`. Since `store` returns `unit`, the value of the `Rpc.rpc` call can be ignored, and the return value of the function is the value of the final statement, which is the special symbol `()` which represents `unit`. Another possible implementation might construct an appropriate marshaling function and pass the function itself to the `Rpc` structure (using the constructor `FN` rather than `MSG`).

```

signature DBSRVSTUB =
sig
  exception NotFound
  val db_server : unit -> unit
end

```

Figure 3.14: Server Stub Signature

```

functor DbSrvStub (structure TType: PTYPES
                  structure Pres: PRESENTATION
                  structure Rpc: RPC
                  structure DbSrv: DB
                  sharing Rpc.Buffer = Pres.Buffer
                  and type TType.T = Pres.T = DbSrv.vtype)
: DBSRVSTUB =
struct
  exception NotFound

  < private declarations and definitions >

  fun db_server () =
    let
      val pid = 1467
      val p = Rpc.create [D rpeer = (H_DEFAULT),
                        rport = (P_ID pid)]

      fun loop () =
        let
          val (replyto, msg) = Rpc.recv_req p
          val resp =
            case msg of
              Rpc.MSG m => dispatch m
            | Rpc.FN f => dispatch (f())
        in
          Rpc.send_resp
            replyto (Rpc.MSG (Pres.end_msg resp))
        end
      in
        loop ()
      end
    end

```

Figure 3.15: Simple Server Loop

The cases of the SML case statement are matched by pattern matching the value of the given expression with the patterns of each case. In the process of making the match, SML will also assign values to temporary variable corresponding to the structure of the matched type. For example, since the Msg type is defined as either a MSG of Buffer.buf or a FN of unit -> Buffer.buf, the case statement here not only distinguished between the two choices but also binds the local variables m or f for use in the corresponding arms of the case statement.

```

fun dispatch msg =
  let
    val mb = Pres.open_msg msg
    val (opn, u) = unmarshalheader mb
  in
    case opn of
      "store" => store mb
    | "delete" => delete mb
    | "lookup" => lookup mb
    | _ => raise NotFound
  end
end

```

Figure 3.16: Server Stub dispatch Routine

```

fun store mb =
  let
    val (STRING k) = Pres.unmarshal mb
    val data = Pres.unmarshal mb
  in
    (DbSrv.store (k, data);
     [Pres.marshal UNIT])
  end
end

```

Figure 3.17: Server Stub store Routine

SML pattern matching can also be used in local variable assignment within a let statement, as in the store routine above. Pres.unmarshal returns a PTYPE, but in cases where the expected value is known we don't really care about the tag but want to be able to directly manipulate the value. Since the first argument of the store function is defined to be a string, we can combine the steps and bind the untagged value to the local variable. This constraint is imposed by the application, and not by the language; strictly speaking, the result should be tested for unexpected values before the variable binding in order to avoid the possibility of runtime errors, but the test has been omitted for simplicity of presentation.

Parameterized Modules	•
First-class Functions	◦
Runtime Type Info	•
Concurrency	•
Exceptions	•
Static Type-checking	•

Figure 3.18: High-level Language Features in ML-RPC

3.4.1 Layering

The RPC case study makes extensive use of layering abstractions. The main goal is flexibility, in several senses: reconfiguration, extensibility, module re-use, and ease of use (transparency to applications).

An important step in designing a customizable system is in identifying a modular breakdown of the functionality. Isolating functionality that is likely to be customized into small- to medium-sized modules with well-defined interfaces greatly reduces the effort required to implement and substitute alternative functionality.

The most crucial functional units in an RPC system are the presentation routines, which convert program data to and from a transmissible format, and the transport routines, which transmit the data. RPC systems may also incorporate various other functional units, such as those providing authentication, security, or reliability functions, or perhaps even some form of application-specific transformation. Although the basic functionality of any RPC package will be fairly constant, there are many reasons for preferring different marshaling strategies, external representations, transport protocols, or security mechanisms for particular applications. Therefore, a partial definition of flexibility for an RPC package is the ability to support multiple implementations of key functional units.

Reconfiguration

Configurability is based primarily on the parameterized module system. The parameterized modules provide a kind of higher-level polymorphism, allowing the module implementations to vary but keeping the interfaces fixed. This approach allows different protocols to be represented as multiple

implementations of the same signature. Because the module system is parameterized, the available set of modules can easily be recombined at link time to provide the desired combination of protocols and features: a particular functor parameter can be assigned different module implementations much as a procedure parameter can be assigned different values. A useful feature of a parameterized module system is the ability to easily modify a functor or structure's name and/or interface by "wrapping" it in a new functor with a new signature: this makes it easy to compensate for slight variations in the interfaces (signatures) of the base modules, increasing the re-usability of the code. Furthermore, since only the signatures are fixed, extending the set of available protocols does not affect the overall design of the system at all.

A system implemented in any programming language can maintain libraries, but for reconfiguration to work properly the exported signature for each alternative implementation must be the same. This can cause naming conflicts in many languages if the different implementations are co-located within the same library or application. The parameterized module system in SML allows a functor or structure implementation to be bound as a parameter to a local identifier within the scope of the functor being instantiated, making it clear which module is being referenced when a particular routine is invoked.

As an example, consider the `Tran` parameter of the `Rpc` functor. The basic transport signature used in the RPC case study is shown in Figure 3.9. Because of its generality, it can be used as the basis for most higher-level transport protocols, such as RPC or message passing. As long as it exports the signature specified by the `Rpc` functor definition, the implementation of the structure assigned to the `Tran` parameter is arbitrary. Furthermore, assuming the same basic functionality, it is straightforward to provide a wrapper structure which converts an existing module signature into the specific signature required. This means that existing packages can easily be adapted for use in the RPC package structure. To model an RPC signature with a message passing structure, for example, one could construct a routine `rpc` which first invoked `Mp.send` and then `Mp.receive`.

For example, one of the transports used in the RPC case study is an SML interface to TCP sockets. The `ML-Sockets` signature exports an interface to a large portion of the UNIX socket system calls. Mapping the functionality to the specified transport signature requires packaging the relevant functionality into the appropriate routines, and effectively eliminating the rest. Consider the following non-blocking receive operation (see Figure 3.19) from a functor exporting the `TRANSPORT` signature and parameterized by a `SOCKET` signature. The function header automatically deconstructs its argument, which is of type `msginfo`. It checks whether there is a message available on the specified socket: if not, it raises the exception `Msg`, and otherwise it invokes the standard socket call `recvfrom`, and uses the return value to construct its own result. Since the Mach IPC signature⁶ also provides the base functionality to construct an implementation of the above interface, it could be transformed in a similar fashion.

Although functor parameters are bound at link time, a certain amount of runtime configuration

⁶implemented by Fritz Knabe.

```

functor TranDSock (structure Peer : PEER
                  structure Sock : SOCKET)
: TRANSPORT =
struct
  type msginfo =
    {conn: Net.fd, address: Net.address option}

    < more functor body >

  fun recvnb {conn = c, address} =
    if (not (Sock.read.ok c)) then raise Msg
    else
      (let
        val (msg, replyto) = Sock.recvfrom c < ... >
      in
        ({conn = c, address = (SOME replyto)}, msg)
      end)

```

Figure 3.19: "Wrapper" Functor for Sockets

is still possible. A stub could import n different communication packages, and allow each call to specify the desired protocol by choosing the appropriate datatype constructor. Alternatively, taking advantage of first-class functions, the application level could import the relevant protocols and assemble each into a record of protocol routines. The appropriate structure could then be chosen dynamically by the application, and passed as an argument to the stub routine⁷. Unlike similar strategies in conventional programming languages, such function records would be as type-safe as any other part of the language.

Extensibility

This particular package provides fixed signatures for the RPC, PRESENTATION, and TRANSPORT modules. It provides implementations of each of these, although the implementations are intended as the basis for a library of alternatives rather than as canonical implementations. The package also provides some support modules for tying together interface types: PEER, for representing the remote peer, BUFFER, for representing the message structure, and PTYPES, for representing the

⁷ Although the total set is still fixed at link time by the application parameter bindings, from the point of view of the stub module the choice is strictly dynamic.

tagged set of transmissible types. A production system would also include a customizable stub generator.

The stub routine is parameterized by structures exporting the `TRANSPORT` and `PRESENTATION` signatures, as well as any other modules desired (e.g., structures exporting encryption and checksum functions, if they exist, or perhaps a lightweight thread package). Since the stub routines coordinate the message processing before and after transmission via the RPC package, they are free to apply whatever transformations they like to the data as long as it is in the correct format for each specified interface. Note that the structure of the code is not targeted for any specific set of modules. The stub generator need only ensure that any particular client/server pair supports identical or complementary encoding and decoding schemes. Adding a new module implementation need only involve those applications actually using it. For example, if a new client wanted to use a new external representation format, only the servers with which it wanted to communicate would need to be modified. Any existing clients would be unaffected by the extension.

Module Re-use

There are of course many other modules used for the various concrete implementations. These can be strictly independent, but since many such designs use the same basic abstractions it makes sense to share the underlying modules wherever possible. This is sensible because it avoids the programmer overhead of re-implementing heavily used program fragments; it also makes it more likely that the shared code will have fewer bugs and be more efficiently implemented than single-use code because of its longer lifetime and wider use. This approach also simplifies the effort required for extensibility, since it is likely that most of the underlying abstractions required for any particular protocol will be available in the system library.

The naming hierarchy imposed by the functors simplifies the re-use of existing code. Implementors often want or need to use existing implementations of various program blocks in their systems. Sometimes the interfaces won't match exactly; sometimes the interfaces will match, but the functionality of a particular routine won't be quite right. This can cause virtually insurmountable problems in an environment where, for example, all procedure names are global, but it is a straightforward task in SML. The programmer could simply implement a new functor exporting the expected interface, and provide the existing (old) module as a parameter. The new functor can then use the old implementation, and any other desired parameters, to produce output in the form required by the caller. Furthermore, the new functor also provides a new level in the naming hierarchy, avoiding any problems with reusing routine names or interfaces.

It is also possible to take an even broader perspective. The client/server model has proven to be a successful paradigm for developing distributed systems, but it is certainly not the only choice. Most communications packages are composed of the same basic functional units as RPC, so there is no reason that the same building blocks can't be configured into any number of other protocols. For example, constructing a message passing protocol need only be a matter of writing the signature

and implementing it as a functor making calls to the appropriate RPC support modules.

In principle, of course, module re-use is not restricted to high-level languages. In practice, however, complications such as adjusting interfaces, coordinating variable names, and deciphering type representations tend to prevent this sort of sharing except within fairly tightly-knit organizational groupings.

Transparency

It is generally agreed that transparency is a priority for most distributed systems. Except for providing a logical handle with which to specify a remote peer, it is generally easier on the programmer if the interaction appears to be local. A common technique is to have a mental model of a distributed system as a non-distributed application with a communication package layered in between the peers. The ability to pass first-class functions, or closures, across an interface makes layering *per se* much less onerous than in conventional languages. A closure may encapsulate an arbitrary amount of "knowledge" about the internals of a particular layer. If this closure is passed to another layer and invoked there, the modularity of the system has not been compromised, yet one achieves the enhanced performance effects of "layer collapsing".

An important benefit of the layering approach is that it encourages modularity of design. When this modularity is further supported by a parameterized module system like that of SML, each layer automatically gets all the protection of type safe encapsulation for all of its internal variables. This means that each module may safely have an independent set of names without worrying about the naming schemes of other modules, and know that the representations and values of its local objects are safe from malicious or accidental modification from outside its scope.

There can be some additional overhead involved in adding layers to a system, but if the mapping is straightforward the compiler will often be able to optimize much of it. For example, if a procedure implementation in one structure simply invokes a corresponding procedure in another structure, the compiler will be able to optimize out the intermediate procedure.

By designing and implementing at a higher level, we have a better opportunity to customize the evaluation strategy to the expected data usage patterns. Our simulations demonstrate that a functional programming language, and in particular the use of first-class functions, provides an attractive environment in which to implement layered communication software. First class functions unify control and data flow, providing a much more flexible interface between program modules than with conventional programming languages. The flexible interface means that the integrity of a module can be maintained without unnecessarily limiting the amount or form of the data that must cross its boundaries.

3.4.2 Evaluation Strategies

The client application has the power to customize features at a per-call granularity. Depending on the high-level design of the stub module, different classes of remote calls could be processed by different security procedures, or an application can choose to use RPC for one operation and message passing for another. Another possibility is to associate a set of marshaling functions with each RPC connection.

To constrain a set of choices, the programmer can define a datatype with a constructor for each supported option. Another solution is to allow for a function argument with a specified interface. For example, the client stub routine could compose an encryption function to the return value provided by the presentation module. SML's facility for constructing anonymous functions dynamically makes it easy to convert any function into a function of no arguments by constructing an anonymous closure on-the-fly. The missing marshaling code (marked by the ellipsis) can be found in Figure 3.13.

```
val pres_fcn = fn () =>
    (E.encrypt (Pres.end_msg
      [Pres.marshal ...
    ]))

< ... >

Rpc.rpc con (FN pres_fcn)

< ... >
```

The SML keyword `fn` is a function specifier, similar to the λ construct in the λ -calculus, which can be used to construct an anonymous function. The above construction assigns the variable `pres_fcn` to a function of no arguments which applies the (predefined) function `E.encrypt` to the result of marshaling the relevant parameters. Assuming that `E.encrypt` takes a `Buffer.buf` and returns a `Buffer.buf` containing the encrypted result, `pres_fcn` can be provided as a function parameter to `Rpc.rpc`.

Functional programming languages and first-class functions also provide support for fine-grained parallelism. Functional programming style encourages small, self-contained functions; closures ensure a well-defined execution environment. A parallelizing compiler should be able to easily exploit this structure, or it could be done more explicitly by the programmer using a lightweight threads package.

Mechanisms like parallel RPC are one way to provide this sort of processing overlap. For cases where the same message needs to be sent to multiple destinations (e.g. for replication, or for distributed Linda tuple-space), providing a call with parallel semantics allows both the client setup processing and the network latency and server computation time to be overlapped. Even if true multicast support is unavailable, this approach can result in significant performance improvements [52].

Another useful feature of SML is partial evaluation of curried functions. This can be particularly useful in the context of an RPC system for keeping connections straight. If there are a few heavily used routines, it is possible to bind the connection parameter of the function, and use the resulting function directly with the remaining arguments. This can reduce programming errors from mis-handling connection identifiers. Curried functions are particularly useful if it makes sense to bind the arguments to values at different times. For example, it could be useful to curry an RPC function which takes a connection identifier and a set of server arguments, since a client would tend to re-use the operation many times with the same connection argument but different server arguments.

3.4.3 Type System

Strong typing is often a point of contention in programming language discussions. To type theorists and language designers it is both necessary and desirable; to some systems programmers, it is the bane of their existence. Since there is some validity to both perspectives, it has proven a difficult issue to resolve.

Strong Static Typing

A strong static type system provides a well-defined, structured programming environment and can greatly simplify the task of the programmer. It can provide the basis for proving theorems about the correctness of programs; a language like SML can provide the framework for work on automatic verification of program correctness. A sufficiently strong static type checking mechanism provides guarantees about runtime type safety even in the absence of runtime type checking; a surprising number of runtime type errors can be eliminated by strong static type-checking. In addition to the usual careless mistakes and typographical errors, complex semantic or conceptual errors often manifest themselves as type errors. Furthermore, enforcing the encapsulation boundaries of abstract types increases the programmer's control over the local state, and therefore over the ultimate behavior of the program.

An advantage of the static mechanism is that it provides strong guarantees without the cost of dynamic (runtime) type checking. Although there are instances, such as the transmission of polymorphic types, where dynamic type information is indispensable, in general it is very advantageous to be able to get such strong type guarantees and only have to pay the static cost, at compile and link time.

Another advantage to a strong type system is that, since the module interface specifications are an integrated part of the language, they can not diverge from the server implementation: they can be reliably used as server specifications for the client and stub generator. The function of the signatures is to fix the interfaces and maintain type safety while allowing evolution of the corresponding implementations. This concept has even more relevance in distributed applications, where it is impractical to assume that physically remote distributed system components will *not* continue to

evolve. The language-integrated signature mechanism provides an unambiguous point of reference for otherwise independent components.

On the other hand, the fact remains that at the lowest level there can be no concept of type safety. At some point, all data is reduced to a sequence of bits. As long as our type-safe languages are built on platforms of unsafe lower-level systems, the need to manipulate the data and environments outside of the type-safe boundaries requires some form of escape hatch from the type system. For example, suppose an SML needs to invoke a UNIX system call. It can carefully construct a type-safe representation of the necessary C types using an array of bytes, but when it receives the return value from the system call it has absolutely no way of safely validating the type or format of the value it receives. At this point, it must simply trust that the value is of the expected form, and coerce it to a corresponding type within the type system of the language.

Systems programmers often find it necessary to communicate across the language boundaries to various parts of the underlying system(s), and this is especially true in a distributed environment. Not only must the language types be transmitted across language boundaries, but they must also be linearized into a form suitable for transmitting across the network.

There is also the philosophical question of the meaning of type equivalence across address spaces. How can we decide if types instantiated in different domains are equivalent? If we accept that base types are equivalent, we can conclude that constructed types with equivalent names and visible, equivalent structures are also equivalent. However, opaque types such as abstract types or functions can not be reliably tested for equivalence.

The strong type systems of high level languages also introduce some complications in a distributed heterogeneous environment. Most existing static type systems can only provide their type guarantees within the domain of a single address space or session. A distributed system necessarily goes beyond those boundaries, but wants to retain the integrity of the type system as well. The type guarantees cannot be absolute without the existence of a distributed implementation of the language itself. Just as threads are coordinated by the top-level loop of the language, true distributed type management would require a meta-top-level coordinator for the language instances composing a particular distributed system.

A strong type system has many benefits, but the combination of a strong type system and polymorphic or abstract types poses a major problem for a communication protocol which requires structural information in order to linearize a typed object for transmission. A stub generator can provide enough static type information to marshal arguments of known type, but the type of a polymorphic parameter can be known only at runtime, and it is a violation of type safety for the representation of an abstract type to be known at all outside of the module in which it was defined. It should be noted, though, that the transmission problems are only with respect to types that are unlikely to be supported at all in conventional programming languages.

Some of that complexity can be side-stepped if we are willing to trust a stub generator to the same degree that we trust the implementation of the language or the operating system. If we can provide a trusted stub generator which uses a signature file and an external representation format as fixed

input, and if we are willing to trust the security and authentication mechanisms of our communication system, then we can extend the type safety guarantees across the language boundaries.

Abstract Types

SML's distinction between interface and implementation provides a great deal of flexibility in type-safe layering and configurability. However, in the domain of remote communication, this same separation causes a problem for abstract types. Within the normal semantics of SML, the implementation of an abstract type is unavailable outside of the module which defines it. However, the opacity of the type representation provides some complex problems for remote communication. The communication system needs to know the type representation in order to marshal a type instance for transmission, but making the representation known outside of a particular implementation defeats the purpose and benefits of an abstract typing mechanism. There is currently no way to transmit abstract types within the current SML system, especially among heterogeneous machines, without either subverting the type system or requiring additional information from the implementation at stub creation time.

We chose to side-step the issue of transmitting abstract types in this implementation: other work has explored solutions to the problem without yielding any completely satisfying solutions. Although the case study does not address this issue in detail, we briefly sketch three approaches to the transmission of abstract types: user-provided marshaling routines, proxy objects, and garbage-collection-based linearization. We then outline an approach combining the first two of these approaches that would provide a simple extension to the existing RPC system.

One method is to require any module implementing a transmissible abstract type to provide explicit marshaling and unmarshaling routines [39], preferably into intermediate-level SML types rather than directly into transmission format. The system would then apply the standard transformations to the explicitly typed result. A variation of this idea is to provide type implementations as well as module signatures to the stub compiler, and let it generate special-purpose marshaling routines for each relevant abstract type. However, this variation has the drawback that, without compiler modifications, there is no way of ensuring that the implementation provided to the stub compiler is the same one that is linked into the final system.

Another approach is to simulate communication of abstract types. Rather than attempting to send the objects, the system sends a unique handle for each one [36]. The handle is recognized as such on the remote site, and any specific references to its structure generate a callback to the originating site. This approach does not subvert the SML type system, but has the drawback that it can cause arbitrary amounts of remote communication in a way that may be non-obvious to the application programmer. In addition, if the motivation for transmitting objects is to allow computation to occur on a remote processor, this approach invalidates the effort by forcing all computation involving abstract types to take place on the machine on which they were defined.

A third approach is to use the SML garbage collector to do the linearization [46]. This has

the advantage that it is relatively efficient and can handle any legal SML type. The disadvantages, however, are that it will only work for homogeneous environments, since the data is treated as opaque monolithic blocks; also, should the type implementations differ on the local and remote machines, the heap on the remote machine may be compromised.

Probably the most reasonable compromise for a heterogeneous environment is a combination of user-provided marshaling routines and proxy objects: this allows the implementor to decide which types need to be transmissible for efficiency reasons. A special stub compiler could be provided which automatically extends a particular type implementation with marshaling and unmarshaling routines conforming to a particular external representation format. This compiler might take the signature and functor implementation as input, and output an augmented functor, parameterized by a fixed presentation interface, which uses the presentation routines to marshal and unmarshal the representation in the functor implementation to and from the external representation format associated with the presentation module parameter. The normal stub generator would assume the existence of this augments, and would automatically assume the existence of the marshal and unmarshal routines in the signature of any structure exporting a transmissible abstract type.

Polymorphism

Polymorphic types also provide a challenge to distributed communication. The problem of accessing runtime type information for polymorphic types was highlighted in the process of implementing the presentation routines for the ML-RPC package. SML has strong static type-checking, and it does not currently keep around any runtime type information. In order to have access to structural information at runtime, it was necessary to define a type consisting of a tagged set of basic types (see Figure 3.10), and use them to tag polymorphic types.

One approach to distributed polymorphism is to take the view that since it is only legal to apply generic operations to polymorphic types, they can be considered opaque from the point of view of the remote peer. Then, the proxy object scheme applied to abstract types can be extended to include polymorphic types as well.

The proxy object approach is not acceptable if distinct instantiations of a particular signature in distinct address spaces are to be treated as the same type. Although this is technically illegal from the point of view of the SML type system, this is a logical extension based on the notion of a distributed system as distributed fragments of a single conceptual system. Although it is possible to get around the problem by constructing type tags and applying them to types as clues for the presentation routines, there is fundamentally no substitute for language supported dynamic type information.

Although the tagged-type approach was used in the case study implementations, it is not recommended. Rather, high-level languages should incorporate the capability to specify that runtime type information should be maintained for designated objects. In fact, there are plans to add a new *dynamic* type to SML/NJ that will fill this function. Implementations of communications software

should not be attempted lightly in strongly typed languages which do not incorporate language support for runtime type information.

3.5 Conclusion

A strong type system introduces some complexities into distributed communication, but it also adds a lot of power in terms of predicting program behavior and providing correctness guarantees. In conventional languages, it is rare to have any reliable correctness guarantees even for non-distributed systems. A strong static type-checking mechanism guarantees runtime type safety for non-distributed programs, and can provide a basis for limited reasoning about distributed correctness as well. This means that programs do not have to pay a price at runtime to ensure correctness. Systems programs in conventional languages often "optimize" away the runtime type checks, but when they do opt for type safety they must pay the price at runtime. The runtime performance price is also relevant in the domain of language supported exceptions versus dynamically checking function return codes at runtime; programming with exceptions does not exact a performance price in the normal case.

We can see that features like language supported encapsulation, when supplemented with strong static type-checking, are particularly well-suited to distributed systems programming where modularity is a reality imposed by the distribution rather than simply good programming practice. The ability to reason about the behavior of modules in isolation from the rest of the system is crucial to effective distribution, and the reliability of encapsulation for reasons of name and data security is essential for any sort of accessible modular reconfiguration.

Distribution also brings to the surface many hidden assumptions about environment sharing. In a truly distributed system, it is impossible to ensure that each distributed component maintains exactly the same environment even within the language domain. For many operations associated with remote communication it is essential to consider the environment as part of the transmissible data; most conventional languages do not even have a notion of environment.

Autonomous administrative domains and the inevitability of independent program evolution make it imperative to have an interface specification mechanism that is fully integrated with the language. The integration provides at least some limited guarantees preventing the interfaces to diverge from the implementations.

The RPC case study illustrates the efficacy of applying high-level language support to the issues involved in constructing a type-safe distributed communication system.

Chapter 4

ML-Linda

4.1 Motivation

The Linda programming model attempts to provide a simple interface to medium- to coarse-grained parallelism and interprocess communication primitives, but at the same time isolate the application programmer from the complexities of systems programming. Linda models data as *tuples*, or ordered collections of typed values, and communication as data movement in and out of an abstract *tuple-space*; this unifies the primitives for data and control by transforming communication issues into a location-independent tuple-matching process.

The Linda case study was chosen to provide an example of a system with the goal of providing applications a simple but powerful interface which hides the complexities of communication and parallelism. The ML-Linda design demonstrates how easily high-level language mechanisms can be used to support this kind of parallelism and programming model. A parameterized module system provides a safe, flexible framework for a natural layering of system modules, making it simple to construct multiple system configurations from a set of basic modules. The `eval` operator, which provides explicit parallelism by evaluating its active tuple argument in a separate process or thread of control, has been problematic in most Linda implementations because of the difficulty of accessing free variables; however, it can easily be supported in a language with closures because free variable bindings are preserved automatically¹. The ML-Linda case study was also intended to illustrate how easily a well-known application such as Linda can be implemented in an advanced programming language, and how much additional implicit flexibility such an implementation can provide.

¹This doesn't adapt to distributed Linda without a function transmission mechanism, of course, but the same is true for an implementation in any language.

4.2 Linda Overview

Linda is a programming language extension which consists of a set of high-level operations that can be added to a base language to yield a parallel dialect of that language [10, 23]. The Linda programming model consists of *tuple* objects, an associative memory called *tuple-space*, and a set of operators which act on tuples: *out*, *eval*, *in*, *rd*, and the predicate operators *inp* and *rdp*. The unit of communication is the *tuple*, an ordered collection of typed values. Tuple elements can be either *formals* or *actuals*. *Formals* match elements of arbitrary value but fixed type; *actuals* specify both a type and a value for the match. There is also a special wildcard formal which matches elements of any type.

All communication and synchronization in the Linda model is accomplished by moving tuples into and out of tuple space. A Linda program selects a tuple by specifying another tuple as a *template* to be matched against the contents of tuple space. If tuple space contains more than one matching tuple, a nondeterministic selection is made².

The *out* and *eval* operators both add tuples to tuple space: *out* adds a *passive* data tuple to tuple-space, and *eval* adds an *active* tuple to tuple space. The *eval* operator introduces explicit parallelism by causing the active tuple's fields, some of which may contain arbitrary expressions, to be computed by an independent process. Once all the fields of the active tuple have been evaluated, the tuple is added to tuple-space as a conventional *passive* tuple.

The *rd* and *in* operators are both used to access tuple-space data. The *rd* operation is similar to a lookup operation: it uses the Linda matching algorithm to match a specified *template* against the contents of tuple-space and returns the result. The *in* operation is a destructive version of *rd*: the matching tuple is removed from tuple-space. The matching algorithm is guaranteed to return a match if one exists; if there is more than one match, the choice is nondeterministic. The elements of the matching tuple are automatically bound to any corresponding formal parameters in the template argument. Both *rd* and *in* are blocking operations; if no match is found, they will remain blocked until a matching tuple is added to tuple-space. The predicates *inp* and *rdp* are nonblocking versions of *in* and *rd*.

Linda has been implemented for several programming languages and hardware platforms, including various multiprocessors as well as workstations on a local area network (LAN).

4.3 Design and Implementation

4.3.1 ML-Linda

The ML-Linda [54] case study shows how the language mechanisms embedded in SML naturally support the Linda implementation and programming model. ML-Linda provides a flexible, ex-

²The Linda tuple matching algorithm is defined in [62].

pressive environment for implementing distributed systems by combining Standard ML's powerful type system and support for functional programming with the Linda model of parallel programming. Linda's shared distributed tuple-space complements the functional programming style by providing a natural mechanism for maintaining shared global state; furthermore, the implementation does not require a preprocessor or any compiler modifications. The system also provides location transparency for tuples, and all necessary synchronization.

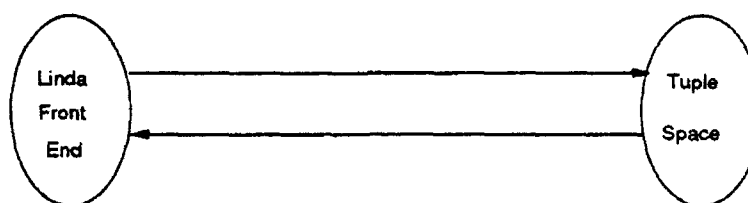


Figure 4.1: Local Linda Configuration

The multiple layers of the implementation provide a framework for exploring issues of flexible configuration, especially when considered in combination with the choices of the communication layer. The main modules in ML-Linda are the Linda front-end, client distribution, communication, server distribution, and tuple-space modules. This modular approach permits the construction of multiple system configurations by different compositions of the individual modules. A local ML-Linda configuration connects the front-end directly to tuple-space (see Figure 4.1); a remote configuration binds the front end and tuple-space modules to the corresponding client and server communication stubs (see Figure 4.2); and a distributed configuration binds the front-end and tuple-space modules to their respective distribution modules which in turn are bound to the appropriate communication stub modules (see Figure 4.3).

4.3.2 Linda Runtime Types

ML-Linda supports the standard Linda operators³ with a slightly modified syntax in order to specify runtime type information for transmission and for the Linda match algorithm. SML uses a strong static typechecking mechanism, and does not maintain runtime type information. In order for the tuple matching algorithm to work, the Linda system must have access to runtime type information on tuple-space fields. Because of SML's current lack of runtime type information, the system uses a SML datatype with constructors functioning as tags⁴ (see Figure 4.4).

The type-tagging datatype `T` supports integers, strings, booleans, lists, and pairwise combinations of these, as well as the SML type `unit`. The `PAIR` constructor functions like a `cons` operator: it

³The syntax of the `in` call had to be changed slightly because of a conflict with the SML keyword.

⁴This results in some limitations in tuple-space field types, but they are mainly related to issues of transmitting abstract types and are problematic in any distributed Linda implementation.

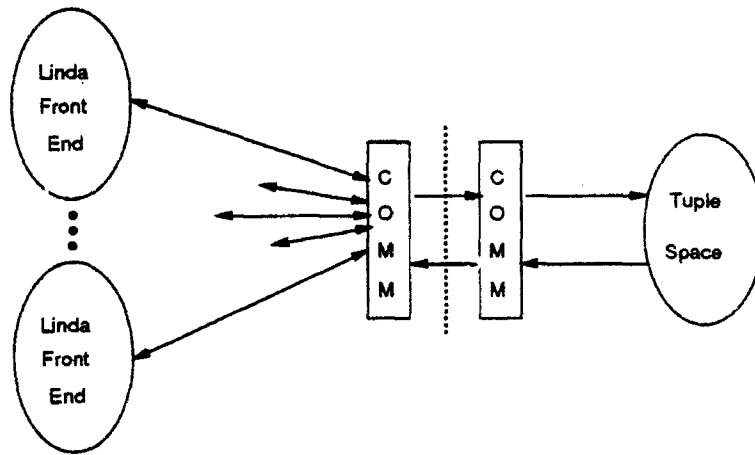


Figure 4.2: Remote Linda Configuration

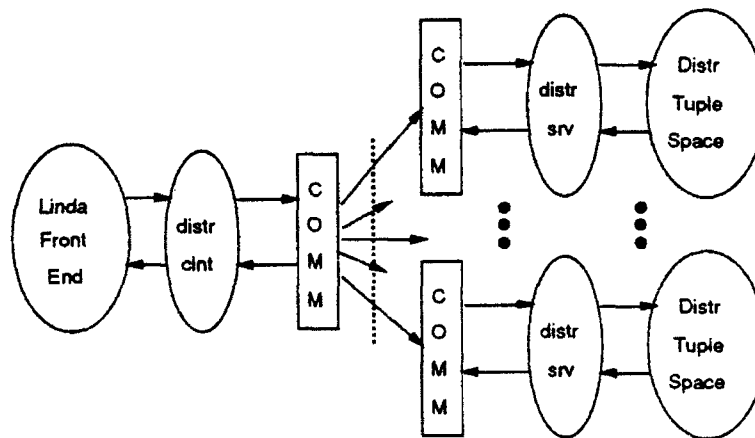


Figure 4.3: Distributed Remote Linda Configuration

```

signature LINDA_TYPES =
  sig
    datatype T = UNIT | INT of int | STRING of string |
      BOOL of bool | PAIR of (T * T) | LIST of T list |
      INT_FORMAL | STRING_FORMAL | BOOL_FORMAL | WILDCARD |
      FN of unit -> T list
  end

```

Figure 4.4: ML-Linda Types

The datatype keyword indicates a complex type composed of a discriminated union of type constructors, which may in turn take arbitrary parameters. The constructors of this particular type are public, since they are enumerated in the signature; for abstract types, the datatype definition is restricted to the implementation, and the type keyword is used to name the type in the signature.

provides the ability to construct complex structures or lists. The `FORMAL` constructors are part of the Linda syntax for specifying a match to an arbitrary value of the specified type; the `WILDCARD` matches an arbitrary value of any legal type. The `FN` constructor, which is used to tag function-valued objects, is restricted to arguments of the `eval` operation⁵. Additional language constructs such as structures, arrays, and lists can also be added explicitly to the union, but the smaller set suffices for the purposes of illustration⁶.

```
[STRING "hello", INT 5, BOOL true, WILDCARD]
```

Figure 4.5: An ML-Linda Tuple

A Linda tuple is represented as a list of these tagged types (see Figure 4.5). A tagged type is constructed by applying the desired constructor to a value of the appropriate type, e.g. `INT 5`. Tuples can be of arbitrary size, making an SML list a natural representation which maintains the necessary properties of Linda tuples and still allows a fixed interface. In addition to a set of basic language types, the special Linda *formal* types and *wildcard* are also supported.

⁵This restriction is due to the inability of the current system to transmit functions.

⁶A forthcoming version of Standard ML of New Jersey will support dynamic (runtime) types; this may allow us to simplify our Linda interface.

```

signature TUPLESPACE =
  sig
    structure LTypes: LINDA_TYPES
    exception Type
    exception NotFound

    val init: string list -> unit
    (* tuple -> unit *)
    val out: LTypes.T list -> unit
    val eval: LTypes.T list -> unit

    (* template -> tuple *)
    val rd: LTypes.T list -> LTypes.T list
    val l_in: LTypes.T list -> LTypes.T list
    val rdp: LTypes.T list -> LTypes.T list
    val inp: LTypes.T list -> LTypes.T list
  end

```

Figure 4.6: Tuple-Space Signature

4.3.3 Linda Front-End

The main part of the ML-Linda interface is the specification of the Linda tuple-space operators (see Figure 4.6). The Linda front-end exports the same interface as tuple-space, so it does not require a separate signature. In fact, all operations but `eval` are forwarded directly to the corresponding operation of its `Tt TupleSpace` parameter.

The `out` operation takes a tuple, or `LTypes.T list`, stores it in tuple-space, and returns `unit`. This operation can be viewed as a data storage operation, or as a message *send* in the communication domain. The first field of a tuple is often used as an addressing tag.

The `eval` operation takes a tuple with some fields optionally containing the special `FN` constructor⁷. The `eval` operator is provided to allow explicit control of parallelism. Its semantics is to create an independent process, or thread of control, which is responsible for evaluating any expressions in the fields of its active tuple argument, and depositing the result into tuple-space. The location of the active tuple should be transparent to the application; the call to `eval` should be non-blocking. Logically the active tuple enters tuple-space immediately, but by Linda semantics it cannot be matched until it is fully evaluated. ML-Linda implements `eval` as a separate thread of

⁷This constructor is illegal except within a tuple provided as an argument to `eval`, because in the absence of a function transmission facility it creates a discrepancy between the local and remote configurations.

control in the address space of the spawning application; this provides the correct semantics in the absence of a remote function transmission facility. The operation forks a thread which constructs a new tuple by substituting for each FN constructor the result obtained by evaluating the corresponding function-valued argument.

The remaining four operators are for accessing tuples, and they all have the same interface: they take as parameter an `LTypes.T` list representing a matching template, and nondeterministically return a tuple from tuple-space which matches it. The differences in the four operators are semantic: the `rd` and `rdp` operators are read-only operators: they do not modify the state of tuple-space. The `in` and `inp` operators remove the matching tuple from tuple-space as well as returning it to the caller. The predicate operators `rdp` and `inp` are non-blocking. If a match is not found, they will raise an exception. The non-predicate versions will block indefinitely until a tuple match is found.

The `eval` operation is implemented by forking a thread on the client to evaluate the active tuple and then output the result to tuple-space (see Figure 4.7).

ML-Linda also exports an initialization routine, taking as argument a string list, which causes tuple-space to initialize its internal state. In the remote or distributed configuration, the argument can be used to specify the remote tuple-space node(s).

4.3.4 Tuple-Space

Tuple-space is a logical associative memory that is shared among a collection of Linda clients. Tuple-space exports the same operators as the Linda front-end (see Figure 4.6), although in the absence of a function transmission mechanism the `eval` operator is implemented strictly as a client-side mechanism.

The main functions of tuple-space are tuple storage and matching. Tuple space is represented in ML-Linda by a hash table of tuples. Since a tuple-space module can have multiple clients and may be multithreaded, the implementation provides appropriate synchronization and locking on shared data structures. Array slots are locked for `out` and `in/inp` operations, but not for `rd/rdp`. This serializes all `out` and `in` operations, and is consistent with the guarantees provided by the official Linda specification.

Tuple matching is the only method of communication in a Linda system, and it is available in both destructive (`in`) and nondestructive (`rd`) forms. The tuple matching routine implements the matching algorithm described in Whiteside [62]. The combination of recursion and SML's internal pattern matching facility with the necessary type information from the datatype constructors makes it quite straightforward to implement.

Figure 4.8 contains the high-level matching algorithm for ML-Linda. The function `match_tuple` breaks the possible combinations of argument representations into three cases:

- Both lists are empty. If we reach this case, either all previous template/tuple pairs have matched, or both the template and tuple started off empty. In either case, we can consider this a match and return `true`.

```

functor LindaFE (structure TS: TUPLESPACE
                  structure SThread: THREAD_SYSTEM)
: TUPLESPACE =
struct
  structure LTypes = TS.Types
  exception Type
  exception NotFound

  val init = TS.init
  val out = TS.out

  fun eval tuple =
    let
      fun xlate elt =
        case elt of
          (FN fcn) => fcn ()
        | lt => lt
    in
      Thread.fork (fn () =>
                    (out (map xlate tup)))
    end

    < more definitions >
end

```

Figure 4.7: ML-Linda Client eval Operation

The operators `init`, `rd`, `l.in`, `inp`, and `rdp` are simply translated directly into the corresponding operators of the `TS` structure parameter. The operations are defined with the keyword `val` rather than `fun` because the definition in this case is simply binding the function named by, for example, `TS.out` to the local identifier `out`.

The `eval` operation first defines a function, `xlate`, which, when applied to a tuple element, checks the element type: if it is type `FN`, it returns the result of evaluating the associated function, and otherwise it simply returns the element itself. The `Thread.fork` operation forks a thread to execute the function supplied as its argument. In this case, the function-valued argument is defined dynamically using the `fn` keyword, which is analogous to the `lambda` keyword in *Lisp*.

The `map` operation is a basic list operator in *SML*. It takes two arguments, a function and a list, and returns a new list for which each element is computed by applying the function to the corresponding element of its list argument. Like most list operators, `map` is a polymorphic function: its list argument is type `'a list`, its function argument is type `'a -> 'b`, and it returns a type `'b list`. The specification can accept any two types for `'a` and `'b`, but enforces the constraints that the translation function input and output match the types of the corresponding lists.

The forked thread will apply the `out` operator to the result of mapping the `xlate` operation to the tuple.

```

(* returns true iff tuple (arg2) matches template (arg1) *)
fun match_tuple (nil, nil) = true
  | match_tuple (f1::t11, f2::t12) =
      if match (f1, f2) then match_tuple (t11, t12)
      else false
  | match_tuple (_, _) = false

```

Figure 4.8: ML-Linda Tuple Matching Algorithm

This code fragment represents the definition of the function `match_tuple`. The function takes two (tupled) tuple arguments. A function definition in SML can be broken into cases if the types of the arguments permit more than a single representation. For example, since the arguments to `match_tuple` are both lists, each can be either an empty list ('[]' or `nil`), or a non-empty list which can be represented in the form `hd::tl` as the head of the list consed to the tail of the list. SML uses the reserved symbol `::` to represent the cons operator.

The three cases are indicated in the function definition by successive function headers separated by the reserved symbol `|`. The reserved symbol `_` is used as a wildcard for the SML pattern matcher, indicating a value which does not affect the match parameters. The pattern match not only separates the arguments into cases, it also binds the values of the specified parts of the patterns to the variable names provided, so they can be referenced in the body of the function.

The pattern matching has two effects. It binds the pattern components to local variables, and also functions as a case statement which directs the control flow to the code segment appropriate to the structure of the argument(s).

- Both lists contain at least one more element. In this case, the match algorithm invokes the routine `match` to compare the head of the template list with the head of the tuple list. If they match, then it recursively invokes `match_tuple` on the tails of the two lists. If it does not, then the template and tuple do not match, and the routine returns false
- Either the template or the tuple is empty, but the other is not. In this case, the tuple and template are unequal in length, so a complete match is impossible: we abandon the match and return false.

The element matching routine is also broken into cases using SML's pattern matching facility (see Figure 4.9). In this routine, the argument representations vary according to the datatype constructors. The rules for the element match are taken directly from Whiteside [62], as mentioned earlier.

4.3.5 ML-Linda Configurations

Simple Linda can be broken down into two major abstractions: the client interface comprising the Linda operations; and tuple-space, which comprises the tuple storage and pattern matching operations. Depending on the desired high-level configuration, we can supply various intermediate modules which provide the necessary management.

Local Configuration

In the ML-Linda local configuration, the front-end and the tuple-space modules are linked together directly as illustrated in Figure 4.1. The configuration is specified at the time the front-end module is instantiated, by supplying a tuple-space structure as a parameter (see Figure 4.10).

Remote Configuration

As an extension to the model, suppose we want to have tuple-space act as a server, and support multiple clients on remote nodes. In that case, we need to introduce a communications layer for both the client and the server. The communications code can of course be integrated directly into the implementation, converting the simple Linda implementation into a remote Linda implementation. However, these modifications make the code unusable for the local case. Instead, suppose that we design a communications module with the same interface as the tuple-space module. We now have a set of modules that can be assembled into either a local or a remote Linda system.

The ML-Linda implementation follows a client-server model, with one or more individual tuple-space nodes acting as remote servers to the local Linda client application. Most of specific details of the communication layer are orthogonal to the ML-Linda implementation; any communication package which supports the basic send and receive operations could be adapted to fit the Linda

```

(* actuals match if both types and values match *)
fun match (INT i1, INT i2) = (i1 = i2)
  | match (STRING s1, STRING s2) = (s1 = s2)
  | match (BOOL b1, BOOL b2) = (b1 = b2)
  | match (LIST l1, LIST l2) = match_tuple l1 l2
  | match (PAIR (t1, t2), PAIR (t3, t4)) =
      (match (t1, t3) andalso (match (t2, t4)))
(* template formal matches with tuple actual *)
| match (INT_FORMAL, INT i) = true
| match (STRING_FORMAL, STRING s) = true
| match (BOOL_FORMAL, BOOL b) = true
| match (WILDCARD, INT i) = true
| match (WILDCARD, STRING s) = true
| match (WILDCARD, BOOL s) = true
| match (WILDCARD, LIST l) = true
| match (WILDCARD, PAIR p) = true
(* template actual matches with tuple formal *)
| match (INT i, INT_FORMAL) = true
| match (STRING s, STRING_FORMAL) = true
| match (BOOL b, BOOL_FORMAL) = true
| match (INT i, WILDCARD) = true
| match (STRING s, WILDCARD) = true
| match (BOOL b, WILDCARD) = true
| match (LIST l, WILDCARD) = true
| match (PAIR p, WILDCARD) = true
(* and pairs of formals don't match *)
| match (_, _) = false

```

Figure 4.9: ML-Linda Tuple Matching Algorithm

```

- structure ts = TupleSpace (structure SThread = ThreadSys
                             structure LTypes = LTypes);
- structure Linda = LindaFE (structure TS = ts
                             structure Ltypes = LTypes
                             structure SThread = ThreadSys);

```

Figure 4.10: ML-Linda Local Instantiation

interface. We use the ML-RPC system described in Chapter 3. The communication structures appear transparent to the application and server code because they export the same interface as the tuple-space model. A functor instantiation for a remote Linda configuration is shown in Figure 4.11.

```

- structure ts = ClientStub (structure SThread = MyThreads
                             structure Rpc = MyRpc
                             structure Pres = MyPres
                             structure Peer = MyPeer
                             structure LTypes = MyTypes);
- structure Linda = LindaFE (structure TS = ts
                             structure Ltypes = MyTypes
                             structure SThread = MyThreads);

```

Figure 4.11: ML-Linda Remote Instantiation

The communication support provides client and server stub structures that hide the communication details from the application code. The stub structures provide routines for marshaling and unmarshaling each of the relevant argument and result types, as well as control structures for making remote procedure calls. The server communication layer also provides a listener loop that waits for incoming requests from the network and forks threads to service them.

Distributed Configuration

Suppose we want to distribute the tuple-space implementation over a variable number of nodes, and make this distribution transparent to the client(s). Again, we have the option of directly modifying the existing modules, or of designing a new module which can be layered into the existing system when desired. The SML module system supports this kind of layering approach naturally; a new layer module can be invoked transparently by an existing module as long as it is defined by the same

signature as the module it is masking. A functor instantiation for a distributed Linda configuration is shown in Figure 4.12.

```
- structure ts = ClientStub (structure SThread = MyThreads
                             structure Rpc = MyRpc
                             structure Pres = MyPres
                             structure Peer = MyPeer
                             structure LTypes = MyTypes);
- structure Dist = Distr (structure SThread = MyThreads
                          structure LTypes = MyTypes
                          structure TupleSpace = ClientStub
                          structure Peer = MyPeer);
- structure Linda = LindaFE (structure TS = Distr
                             structure Ltypes = MyTypes
                             structure SThread = MyThreads);
```

Figure 4.12: ML-Linda Distributed Instantiation

4.3.6 Distributed Tuple-Space

Distributed tuple-space is a single logical associative memory that is implemented as a set of distinct tuple-space servers distributed over a collection of physically separate nodes (see Figure 4.3). The tuple storage and matching systems are replicated on each node of distributed tuple space, although the stored contents are not replicated. Each node of a distributed tuple-space manages its own resources and exports all of the Linda functionality. All of the logic involved in combining the individual nodes into a single logical tuple-space is located in the distribution module, which is layered transparently between the Linda runtime and its communication layer.

Incorporating distribution into the Linda system creates several implementation issues. One of the most obvious has to do with the placement and access of remote tuples. There are two basic approaches to this: one which maintains location transparency, and one which doesn't. An important goal of the distributed Linda implementation is transparency. The client application should depend only on the interface to tuple-space, not on any implementation-specific aspects having to do with communication, distribution, or physical location. This means that the distribution of tuples in distributed tuple-space must be independent of implementation choices such as the hashing algorithm. Tuples may reside on any valid node of distributed tuple-space (see Figure 4.13), but the client application sees only a single unified view of tuple-space. Ideally, the tuples end up distributed over the available nodes in such a way as to optimize both the necessary network traffic and balance

the loads on the relevant nodes. The transparency also means that the implementation will be insensitive to any tuple migration strategies that might be implemented. The initial implementation takes a simple approach and selects destination nodes by cycling through the tuple-space nodes.

The complex state management issues of distributed tuple-space are implemented in independent modules which can be layered into the Linda configuration whenever distribution is desired. In particular, the ML-Linda client and server distribution layers manage state associated with the multiple nodes of distributed tuple-space⁸.

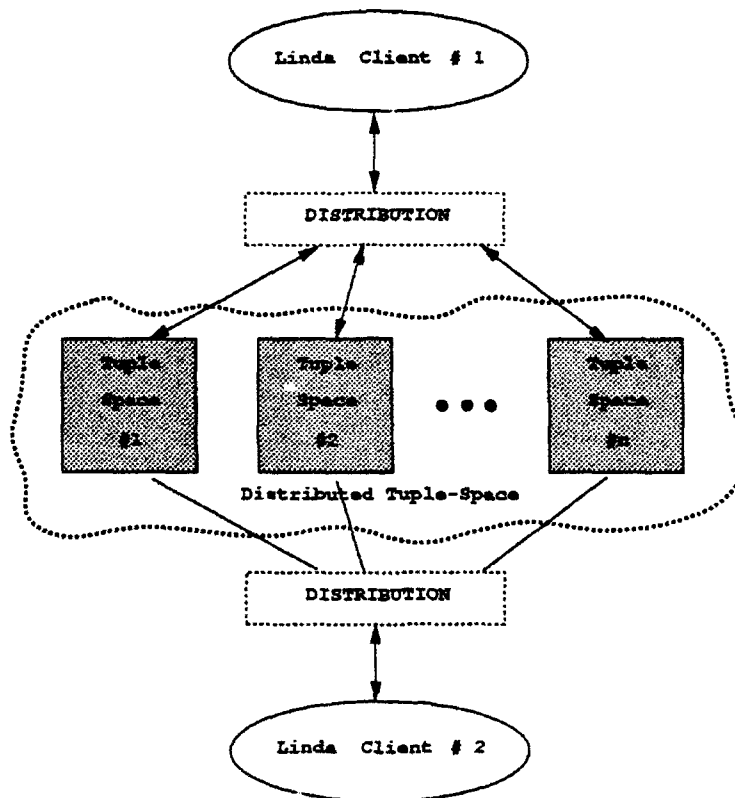


Figure 4.13: Multiple Clients Using Distributed Linda Tuple-Space

⁸Note that the client and server distribution modules are complementary: a system must be configured to use both or neither.

Client Distribution Module

The Linda client distribution module provides a transparent interface to a distributed tuple-space. It is layered between the Linda front-end and the communications package. It exports the tuple-space interface of Figure 4.6, and transforms each Linda request into the appropriate set of remote calls to various tuple-space nodes, manages the necessary state, coordinates responses, and finally returns any result to its client, the Linda front-end.

To maintain the location-transparent view of tuple-space, the communication algorithm must guarantee to return a match if one exists anywhere in tuple-space. The simplistic way to do this is to broadcast all communication to the full set of tuple-space nodes. Replication strategies (see Section 2.2) can provide a cleaner and more efficient approach. Although a replication scheme can normally use any legal combination of read and write quorums (see Section 2), a Linda implementation can only safely use read-one-write-all or read-all-write-one quorums without compromising the location transparency of the system.⁹ This peculiarity is due to the ambiguity of the `rd` operation. Rather than reading a single logical object, the `rd` operation returns any member of an arbitrarily large set of possible matching tuples. Since a set of n replies is not guaranteed to include n copies of the same object, an arbitrary read quorum would not necessarily provide the required overlap with the corresponding write quorum.

The ML-Linda distribution layer uses a read-all-write-one approach to communicate with distributed tuple-space. This means that `out` sends a tuple to a single tuple-space node, while `in` and `rd` request matches from all nodes. We simulate multicast communication by using multiple threads to make the remote procedure calls in parallel. We chose read-all-write-one semantics over read-one-write-all in order to minimize the amount of data transmitted: only one tuple needs to be transmitted for each operation¹⁰. Although `rd` and `in` are sent to all tuple-space nodes, only one match is chosen; furthermore, it is likely that only one version of tuple space will be modified since requests are likely to target a specific tuple. If we used a read-one-write-all scheme, the `in/inp` operation would be more expensive: although the `lookup` operation would be performed on a single node, all of the remaining nodes would have to be involved in order to delete the tuple from tuple-space.

In our distributed tuple-space, with read-all-write-one semantics, an application can receive several different tuples in response to an `in/inp` or a `rd` operation. Although `in/inp` is logically a single operation, it is executed in two distinct phases which must be executed atomically. The `rmv` phase of the `in` operation must tentatively remove the matched tuple from tuple-space, to make it unavailable to any subsequent match requests. Since `in/inp` is a destructive operation, only one

⁹The $n \times n$ processor grid discussed by Bjornson *et al.* [8] allows a valid intermediate quorum assignment that depends on the physical configuration of the system. For this configuration, the write quorum for node i consists of the n nodes in i 's row, and the read quorum consists of the n nodes in i 's column.

¹⁰We actually have `rd` return the matching tuple as an optimization, since we generally expect only a single match and `rd` does not require a second phase for any nodes returning `NotFound`.

match can be accepted; the tentative removals associated with any other responses must be undone. In addition to restoring unwanted matches, it is also necessary to terminate any remote threads that are still blocked before the operation may be safely terminated.

Server Distribution Module

The server distribution module provides a layer of distributed state management around a standard local tuple-space implementation. It exports an extended version of the standard tuple-space interface, breaking some of the operations into pieces to allow for remote commit protocols, and incorporating additional arguments for distributed state management.

Figure 4.14 shows a piece of the `SRV_DISTR` signature. The `init` operation appears unchanged in the tuple-space signature. Distribution does not affect its outcome. The `out` operation also remains largely unchanged: there is no ambiguity in adding a tuple to tuple-space since there are no restrictions on multiple copies of tuples. It is only necessary to add a uniquifier argument in order to preserve at-most-once semantics.

```
signature SRV_DISTR =
  sig
    structure LTypes: LINDA_TYPES
    exception NotFound

    val init : string list -> unit
    val out: string * LTypes.T list -> unit

    < ... >
```

Figure 4.14: Signature for Server Distribution Module `init` and `out` Operations

The `rd` and `rdp` operations are differentiated by whether the operation is to be blocking or non-blocking. Since in the distributed case the blocking operation may need threads or processes killed or state adjusted when a match is found at some node of tuple-space, a `rd` operation is terminated by a `rd.done` operation, with an argument specifying the operation uniquifier. These procedure declarations are shown in Figure 4.15.

The first part of the `l.in` operation is both destructive and potentially blocking. When a match is found, the matching tuple is conditionally removed from tuple-space pending confirmation from the client. Since in the distributed case more than one tuple-space node may return a match, a second phase is required in to commit or abort the pending removal. The clean-up operation is split into two operations, `restore` and `purge`, which correspond to the two possibilities (see Figure 4.16).

```

(* RD/RDP: attempt to find a match for a template.
 * parameters: uniquifier, template
 *)
val rd: string * LTypes.T list
    -> LTypes.T list
val rdp: string * LTypes.T list
    -> LTypes.T list
(* RD_DONE: clean up state for blocking rd operation
 * parameters: uniquifier
 *)
val rd_done : string -> unit

```

Figure 4.15: Server Distribution Module rd, rdp and rd_done Operations

4.4 Discussion

4.4.1 Layering

The ML-Linda implementation illustrates the configuration flexibility inherent in a modular, layered design combined with the power of a type-safe parameterized module system. The communication stub layers insulate the system from the network. This transparent layered approach allows the ML-Linda system to be configured at link-time simply by deciding which modules to specify as functor parameters at functor instantiation. The specification of distributed tuple-space nodes, should the configuration require it, is performed at runtime using the `init` operation. The system can also be expanded to use multiple tuple-space environments [24] by instantiating the desired number of Linda runtime modules. While in this implementation a tuple-space is not a first class type, first-class functions allow them to be operated on hierarchically using the `eval` operator.

```

eval [STRING "mixed_ts",
      TS1.out [STRING "foo"], TS2.rd [INT_FORMAL]]

```

4.4.2 Evaluation Strategies

The Linda model itself addresses the issue of alternative evaluation strategies with its `eval` operator for managing parallelism. Extending the `eval` semantics with closures strengthens the correctness of the programming model and increases the range of acceptable expressions for `eval` arguments. In

addition, the existence of first-class functions permits `eval`'s active tuple fields to be incorporated into the type-safe environment as valid tuple element types.

4.4.3 Type System

Run-time typing is particularly important in Linda, since the most fundamental Linda operation is tuple-matching. It is critical that the integrity of this type information not be lost in the process of distribution or reconfiguration¹¹ Our implementation of distributed Linda requires transparent communication of complex objects and some form of associated runtime type information from one node to another. A `rd` or `in` operation binds the fields of a matching tuple from tuple-space to a set of local variables that can then be modified or referenced by the program, so the type information used in the tuple matching must also correspond to the language type system.

The Linda model does not provide any protection mechanism for tuples in tuple-space. ML-Linda implements the basic Linda model and does not address these issues.

4.5 Conclusion

The immediate goal for ML-Linda was to build an implementation of Linda in an advanced programming language that could be transparently operated in any of several modes: locally, remotely with a single tuple-space node, or remotely with multiple nodes functioning as a distributed tuple-space. The second and third options result in a distributed Linda system, where the client and server processes may reside on separate nodes and communicate via remote procedure call. The client application depends only on the interface to tuple-space, not on any implementation specific aspects having to do with communication, distribution, or physical location. The type-safe encapsulation provided by the parameterized module system allows these restrictions to be implemented simply and naturally, and at the same time makes them enforceable by the language. It also supports a natural layered design and implementation that is naturally flexible and therefore configurable. Furthermore, the availability of support for first-class functions allows the semantics of the Linda `eval` operator to be supported by the language.

The ML-Linda implementation demonstrates the feasibility of combining a functional programming style with the Linda model of parallel programming in distributed tuple-space. The ease of decomposing the Linda system into modules reinforces the suitability of a language-supported

¹¹ Although we do not go into this in detail, the issues of abstract data type transmission apply in the context of ML-Linda just as they do in the context of remote communication in general. In order to transmit an abstract type it is necessary to provide the communication system with information about the implementation that is not available in the signature; this effectively makes the type explicit, and requires a new mechanism such as a preprocessor to extract the relevant type information and make it available to the communication system. Complex types whose implementations are included in the signature are already explicit, and can be readily transmitted. In this chapter we consider only issues involving explicitly defined complex types.

parameterized module system for constructing complex distributed programs. The addition of Linda shared tuple-space complements the functional style and provides a flexible environment with the benefits of both programming models for the development of distributed systems.

```

(* RMV: match a template and remove the matching tuple
 * parameters: key, uniquifier, template, block
 *)
val l_in: string * LTypes.T list
    -> LTypes.T list
val inp: string * LTypes.T list
    -> LTypes.T list
(* PURGE: purge matched tuple and clean up state from rmv
 * parameters: uniquifier, block
 *)
val purge: string * bool -> unit
(* RESTORE: restore matched tuple to tuple space;
 * parameters: uniquifier
 *)
val restore : string -> unit

```

Figure 4.16: Server Distribution Module l.in, inp, purge and restore Operations

Parameterized Modules	•
First-class Functions	•
Runtime Type Info	•
Concurrency	•
Exceptions	•
Static Type-checking	•

Figure 4.17: High-level Language Features in ML-Linda

Chapter 5

Protocol Processing

5.1 Motivation

The knowledge we have accumulated about conventional systems programming is not always directly transferable to distributed systems. Distributed systems have different problems and failure modes than non-distributed systems. They rely on remote communication, which generally increases latencies. The computer science community has begun to assemble a set of new tools, or new combinations of existing tools, with which to build distributed systems. Development of programming abstractions such as replication, concurrency, group communication, and complex distributed agreement protocols have become increasingly important. Because of all the overheads involved, enhanced performance is much more likely to be derived from clever algorithms and program design than from optimizing simple straight-line paths. Providing systems programmers with the proper language mechanisms and programming abstractions is crucial both to the efficiency and to the correctness of the systems they produce.

The increase in network bandwidth means that it is often the performance of the system software responsible for network processing that creates a bottleneck, limiting the bandwidth available at the application level [12, 14, 21]. Some approaches try to ease the bottleneck by parallelizing communications, offloading all or part of the transport processing to an outboard processor [19, 31, 34]. In our increasingly heterogeneous environments, however, the bottleneck is often not in the transport but rather in the presentation processing: linearizing and translating a message to an appropriate format for transmission requires accessing and copying every byte of data.

Messages containing application data have traditionally been broken down into packets at the lower levels of the network hierarchy without regard to any high-level structure. This is a beneficial strategy from the point of view of optimal utilization of packet space, but it has the drawback that, despite their fragmentation, messages can only be processed as atomic units. For example, processing on the receiver cannot begin until an entire message has been received and reconstructed.

and may sometimes wait round-trip network times for retries of dropped packets or packets with failed checksums. This provides little flexibility in addressing the increasing bottleneck in network processing; in order for the system software to keep up with the increasing speed and bandwidth of networks, incoming packets need to be processed as efficiently as possible.

5.2 Overview

Network processing has typically been implemented in a hierarchy of layers [65] which sequentially manipulate the data packets, where each layer represents a different level of abstraction of the data being transmitted. This results in a clean logical model which provides a modular, comprehensible representation of the desired system behavior; unfortunately, the corresponding implementations can be inefficient [14]. In traditional imperative languages, the inefficiencies can only be addressed at the cost of modularity: the engineering model can be restructured and the various processing functions can be regrouped in a more efficient way, but such restructuring violates the layer boundaries. In high-level languages, however, we can achieve similar performance improvements without losing the benefits of the logical layered model by taking advantage of language features such as abstract data types, polymorphism, and first-class functions.

This issue is addressed in the domain of next-generation protocol design by Application Level Framing (ALF) and Integrated Layer Processing (ILP) strategies [14] (see section 2.2). ALF fragments messages into *frames* meaningful to the application, allowing the receiver to process each frame independently; thus, arriving frames can be passed directly to the higher protocol layers rather than waiting for complete message reassembly. ILP bypasses layering boundaries to group related operations on the data, thereby reducing overall processing time.

These strategies are relevant not only specifically at the network protocol level as in ALF and ILP, but also at other levels of distributed systems. Within the a high-level language framework providing first-class functions, closures, and a parameterized module system, it is possible to integrate processing from different system layers while still enforcing the modularity guarantees of each. A strategy which exports encapsulated functionality (closures) as parameters or return values preserves module boundaries but at the same time allows re-ordering and composition of functionality from external modules. This increases flexibility, but nevertheless keeps the logical and engineering models of the system synchronized. The simpler logical model, in turn, requires less programmer effort for implementation or modification.

The third case study is of an implementation of a protocol architecture framework, focusing on the transport and presentation layers, which is used to prototype a set of communication processing strategies. It demonstrates the utility of high-level language support for facilitating rapid prototyping as well as for expressing and evaluating designs and implementations of communication protocol architectures, exploring the ideas behind ALF and ILP in an environment with advanced programming language support. The ALF analog deconstructs argument lists at the sender, and reconstructs them

on the receiver. The ILP analog accumulates functions from different layers into closures, which can be transmitted across module boundaries as parameters or return values and executed without compromising the encapsulation of each layer boundary. This approach also permits the application of a variety of alternate evaluation strategies to enhance program performance by composing related operations or by removing computation from the critical path at the network/system boundary, delaying or possibly avoiding altogether the execution of much of the protocol processing.

As discussed in the previous chapters, layering is a natural, intuitive model for visualizing many kinds of distributed systems. As evidenced by the ILP proposal, however, the immediate efficiency gains made possible by operation reordering provide one reason for considering alternate engineering solutions. Furthermore, such alternatives are also often better suited to the increasingly complex algorithms which take advantage of the potential concurrency introduced by distribution. Modularity of processing as well as of structure increases the effectiveness of parallelism, and of associated strategies such as delayed evaluation.

5.3 Design and Implementation

A diagram of the high-level structure of the protocol processing framework is shown in Figure 5.1. The system consists of a Frame abstraction which encapsulates the frame representation and a set of operations for constructing and extracting frame headers; client and server frame management layers which control the fragmentation and reconstruction of message data, the ordering of protocol operations, and any relevant state management; the application level client and server used to demonstrate the package; and an independent communication layer.

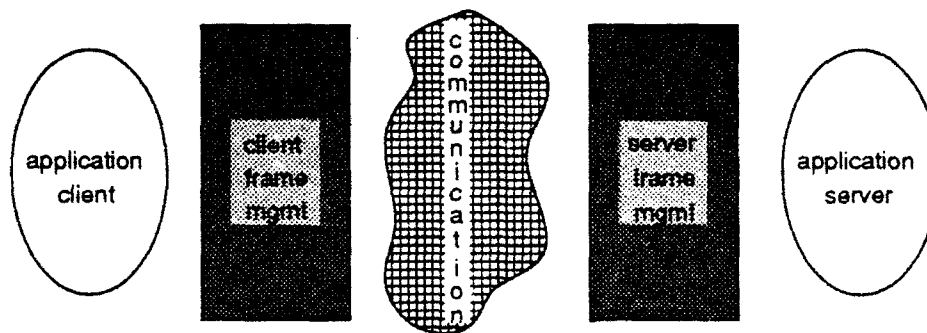


Figure 5.1: Protocol Processing Framework Overview

Using closures, it is possible to specify the control flow of a program dynamically by ordering the desired operations and combining them into a function-valued argument to be passed to the appropriate subroutine. Although others have considered regrouping the ordering of protocol

processing operations to improve overall performance, it has always been by blurring the modularity of the layering boundaries. This blurring effectively complicates the program by distancing the implementation from the logical model; it also increases the interdependence of system components which would otherwise be independent enough to evolve separately. The case study shows that passing first-class function arguments across the layering boundaries preserves the encapsulation of the implementation, but nevertheless allows reordering and composition of operations.

Specifically, each logical layer of the protocol stack is encapsulated in a parameterized module. Inter-module control flow is managed with the support of first-class functions and closures. The implementation demonstrates that logical composition of protocol processing operations can be accomplished by passing first-class functions across the interfaces between modules, and provides a test-bed for examining the effects of various parameters on the protocol processing performance. Although modularity and efficiency are both desirable properties, applying this mechanism in a conventional programming language environment has the unfortunate effect of putting them in opposition: combining functionality from logically separate program layers in order to improve efficiency generally erodes away the corresponding module boundaries. The case study uses high-level language support to extend the module (layer) interfaces to pass functions rather than values, allowing modular function composition across type-safe module boundaries. This yields a clean, modular specification yet allows enough flexibility for more efficient implementations.

5.3.1 Runtime Types

As with the ML-RPC and ML-Linda implementations, we compensate for the lack of runtime type information by using a discriminated union type which supports recursive combination of integers, real, strings, booleans, tuples, lists, and arrays (see Figure 5.2). The transmitted data is self-defining, although the layered design allows alternative implementations (e.g. one providing templates for marshaling and unmarshaling). Depending on the implementation of the marshaling/unmarshaling routines, it may only be necessary for the application to supply the type information for polymorphic types.

5.3.2 Communication Layer Composition

We use the RPC system described in Chapter 3 as the framework for the communication system. Layer interfaces which do not directly support first-class functions are modified with wrapper functors.

As a result, a set of functionality can be enclosed in a closure and passed up or down through the later interfaces to provide the desired execution behavior. Consider an example of lazy evaluation. Rather than fully processing each packet as it arrives, the receiver can create a closure to perform each required function, and associate these with the unprocessed message. For example, a checksum closure can be constructed at the transport level: this automatically associates the relevant

```

signature PTYPES =
  sig
    exception Type
    datatype T = UNIT | INT of int | STRING of string
                | BOOL of bool | REAL of real | LIST of T list
                | ARRAY of T array
  end

```

Figure 5.2: Transmissible Types

environment bindings with the function code for later evaluation (see Figures 5.3 and 5.4). Here the transport routine `recv` invokes the checksum routine `rmv.checksum`, which sets up and returns a closure that will extract the message checksum and compare it with the result of the checksum computation over the message. None of the checksum computation will occur until and unless the closure is invoked, and a failed checksum will result in the `Checksum` exception being raised.

```

fun recv (net) =
  let
    val msg = Net.receive net
  in
    (msg, CS.rmv_cks msg)
  end

```

Figure 5.3: Receive A Message

```

fun rmv_cks pkt =
  ( fu () =>
    if ((unmarshal_checksum pkt) = (checksum pkt))
    then ()
    else raise Checksum)

```

Figure 5.4: Set Up Message Checksum Closures

Returning this closure to the appropriate presentation routines enables the checksum routine to

be applied during the presentation processing even though it may require access to variables internal to the transport or checksum modules. The layer boundaries are strictly enforced, but the relevant information is still available at execution time. This approach serves both to remove the checksum from the critical path and, depending on the implementation, possibly to reduce the total number of data accesses required for the processing by applying the checksum and presentation processing routines incrementally during the same pass over the data.

Alternatively, the application level could provide presentation closures as arguments to the lower levels, allowing all of the processing to execute eagerly at, for example, the transport level.

5.3.3 Frames

A frame is an abstraction of a data packet in which the data has meaning at the application level; logically, it represents a meaningful fragment of an application message. A frame belongs to a specific sequence (message) of a specific connection. The fragmentation and reconstruction strategy for a particular message is restricted only in that each frame must contain data corresponding to one or more application-level types, and that it is expected to specify a message uniquifier, a frame sequence number and an end-of-message flag for each frame (see Figure 5.5). This information is encapsulated in the frame header prepended to each frame; it is used by the receiver to reconstruct the message from the individual frames and to provide information on data placement.

```
frameheader:
    operation_id    : int
    msg_id          : int
    frame_id        : int
    end_of_message  : bool
```

Figure 5.5: Frame Header

5.3.4 Frame Management

One of the objectives of ALF is to make the frame-based system independent of the order in which frames are sent or received. There are many reasons for incoming packets to be processed out-of-order on the receiver: they could have arrived out-of-order because of network resequencing due to lost or corrupted data, or they could be effectively reordered due to the behavior of more complex high-level strategies such as concurrency or delayed evaluation on the receiver, or concurrent processing on the sender.

One of the major advantages of a frame based communication system is that the individual frames of a message are no longer interdependent. This allows more rapid and efficient protocol processing,

since frames can be processed as they arrive rather than in message-sized bursts. Some applications, such as file transfer and video applications, may also be able to take immediate advantage of the availability of the partially reassembled data; however, this is likely to require that the application incorporate knowledge of the transport strategy.

A more general approach is to add a frame management layer between the communication layer and the application which encapsulates the frame-specific knowledge. Such a layer can take advantage of the properties of application level framing and the support of a parameterized module system and first class functions to invoke the server application with the equivalent of *futures* in place of actual arguments. In this case, the computation encompassed in the future construct is to block on the arrival of the data and then perform the requisite protocol processing and unmarshaling. Thus, the system provides dynamic reconstruction of logical messages (frames).

The use of closures allows the possibility of customizing the specific part of an evaluation to delay. For example, in the domain of protocol processing, we might consider whether to delay the checksum operation as well as the unmarshaling of received data. If the checksum is performed immediately on the arrival of a frame, then some of the benefits of the integrated layer processing approach are lost: the data will have to be touched at least once for the checksum operation and again later for the marshaling and unmarshaling. However, if the checksum operation is included in the closure and delayed until some unspecified future time, then the system will need to be able to handle the possibility of a retransmission request generated by a failed checksum.

This can in fact be an application-specific choice. For some applications, the strategy for handling a failed checksum might be simply to discard the packet and continue. In this case, there is no extra management overhead required to maintain old state on the sender, and there is the savings of being able to compose the checksum and the marshaling into a single pass over the data. For an application requiring retries for failed checksums, the situation is different. In this case, the sender must either keep a database of old frames until it receives an acknowledgment of receipt from the receiver; or, alternatively it must be prepared to regenerate any given frame on request in an idempotent manner. There can of course be various combinations of these strategies; for example, the sender could expire frames from its database according to a preset timer. Any retry requests from the receiver arriving after a frame's expiration would be rejected.

To simplify the implementation, we separate the checksum calculation from the marshaling and unmarshaling computation. The checksum is performed by the transport layer on receipt of the frame, and the unmarshaling is encapsulated into a closure and passed along to the application. A more detailed implementation might allow variations of the timing of the checksum calculation and of the retry strategies used by both the sender and the receiver.

Implementation of application level framing would require modifications at the transport layer and below. Since with the current SML platform we do not yet have access to these levels, our implementation is essentially a simulation. Rather than sending the frames as pieces of a single transport-level message, in the simulation each frame is sent as a separate message. To compensate for this, the communication interface for which RPC stubs are generated consists of a pair of routines,

```

val send_frame:  frame -> unit
val recv_frame: unit ->  frame

```

Figure 5.6: Frame Transmission Functions

`send_frame` and `recv_frame` (see Figure 5.6). `send_frame` takes an argument of type `frame` and returns `unit`, and `recv_frame` takes `unit` and returns a reply of type `frame`. For both the client and server, it is the frame management layer which translates the actual application interface to and from the frame-based interface.

Client Frame Management

The major functions of the client frame management are to break a high-level message into a sequence of application-level frames and to maintain sufficient state to respond to frame retry requests. The only formal restriction is that each frame must correspond to a type, or unit of data, defined at the application (or language) level. The degree of fractioning is arbitrary.

One of the interesting issues in considering an application-level framing approach is how to choose the fragmentation strategy. Fragmentation can be chosen statically for all applications; it can be chosen heuristically per application (i.e. coded into the application client and server frame management); or it can be chosen dynamically at runtime, possibly necessitating a negotiation process between the client and server management routines in order to regulate the process of message reconstruction. The simplest strategy to implement for the prototype is a static fragmentation which fragments a message at parameter granularity. This simplifies the fragmentation process for the client, and obviates the need to include a negotiation scheme in the initial prototype.

Frames may be stored in case of retry requests. The entries could be removed lazily after a specified timeout has expired, on demand after acknowledgment from the receiver, or via some combination strategy. The protocol must ensure that the frame database will not cause a storage leak in the case of lost acknowledgments.

Server Frame Management

The server implementation is multi-threaded. Processing begins with the first received frame of any request, and is only interrupted if there is an attempt to access data which has not yet arrived. This is accomplished by invoking the server routine with *futures*, which are effectively closures for accessing the relevant arguments, rather than directly with the arguments themselves. To facilitate the reconstruction process we provide a module which incrementally reconstructs parameter lists from discrete frames, constructing a list of access functions on the arrival of the first fragment. An

element is represented either by a value or by a function which produces a value when invoked. Furthermore, since an attempted access can fail if the requested fragment is not yet available, the access routines provide both blocking and non-blocking options to the caller.

The main part of the work on the receiver is related to the processing and final placement of the received frames in the application address space. Since the frames sent by the client are not constrained to arrive in any particular order, the server stubs must be able to reconstruct complete parameter lists from a sequence of unconnected messages. As each frame is received, it is processed and stored in the appropriate slot in a server data structure. This structure is used as a database of current message fragments; once a server application routine completes, any existing frames of the corresponding message are removed, and any new arrivals will be ignored.

The high-level fragmentation inherent in ALF will result in some wasted space in the physical layer packets, but in return will provide a greater degree of fault tolerance for the processing of lost or unordered packets at the receiver. The additional cost in network bandwidth is proportional to the number of frames times the frame header size; any additional bandwidth is due to wasted space in the packet buffer, and depends on the relative sizes of the frames and the buffer. Although concurrency introduces its own complexities, the ability to fully specify the execution environment makes it easier and safer to perform operations such as parameter marshaling concurrently.

There are several options for ALF presentation processing strategies. Packets can be eagerly unmarshaled as they arrive, and the resulting argument values stored and assembled for later server invocation. They can be lazily evaluated on demand only, passing unmarshaling closures to the server functions to be invoked when and if the parameters need to be accessed. They can be lazily unmarshaled using a *future* construct [33], which introduces concurrent evaluation.

A more interesting possibility than any one of these approaches is to take advantage of SML's abstraction mechanisms to present a uniform argument stream interface to the server application functions, hiding all the evaluation and assembly implementation details, and allow access to several evaluation strategies. All of these options can be expressed naturally in SML; the ability to experiment with these alternative orders of evaluation in a type-safe, modular way is one of the main advantages of this approach.

5.3.5 Application Client and Server

In principle, the structure of the underlying communication system should be transparent to the client and server application processes even when it uses strategies such as ALF or delayed evaluation. The degree to which this is actually true depends on two main points: the amount of control desired by the application, and the amount of support in the implementation environment for the data and control structures required to transparently construct the system components.

In the context of RPC, we have seen that in order to make the communication layer completely (syntactically) transparent to the application it is necessary to leave all connection related choices and

specifications to the system¹. Depending on the application, it may be preferable or even necessary to manipulate connection parameters explicitly. Similarly, various data and control constructs can be implemented more or less elegantly depending on the support provided by the implementation environment. For an implementation which incorporates delayed evaluation of parameters, an environment which supports a future construct can leave the evaluation strategy transparent to the application; with less underlying support, however, it might be necessary for the application to accept a modified syntax in order to use the alternate strategy.

5.4 Discussion

The combination of first-class functions, encapsulation, polymorphism, and exceptions provides the base support for various alternate evaluation strategies. Closures provide a clean mechanism for specifying both the function body and its relevant environment; furthermore, their first-class status allows them to be easily manipulated and accessed. Polymorphism allows the high-level abstractions to be general enough to be applied across a wide range of applications. Exception handling facilities allow the propagation of enough error information to make error conditions meaningful even when computation may take place in a non-intuitive order.

Parameterized Modules	•
First-class Functions	•
Runtime Type Info	•
Concurrency	•
Exceptions	•
Static Type-checking	•

Figure 5.7: High-level Language Features in Protocol Framework

¹The more complex failure modes of remote communication make it impossible to achieve complete semantic transparency

5.4.1 Layering

Encapsulation

The protocol framework demonstrates that logical reordering of protocol layers can be accomplished by passing first-class functions across the interfaces between layers, and provides a test-bed for examining the effects of various parameters on the protocol processing performance.

5.4.2 Evaluation Strategies

A complication of using a delayed or lazy evaluation strategy for the unmarshaling at the receiver is that the sender needs a mechanism for saving or regenerating frames in order to respond to delayed retry requests from the receiver. If packets are saved rather than regenerated, then they will need to be garbage collected once the relevant call completes (with implicit or explicit acknowledgment) or a specified timeout expires. The receiver could be required to verify all packets within the timeout period, or the sender could have the capability to (re)construct packets on demand. The latter strategy is also useful in the context of lazy transmission: the client can transmit small fragments of the parameter data with the request, and transmit further data either lazily or on demand.

All information about locating and extracting the relevant application frames remains internal to the appropriate modules, yet the application is free to begin its processing immediately after the receipt of the first application frame. This not only increases flexibility, but has potential for improving performance as well. As long as the application doesn't actually try to access data that has not yet arrived, it can begin its processing in parallel with any further data transmission. Thus, depending on the amount and complexity of the data transmitted, the processing required on the sender, the transmission itself, and the processing on the receiver can all potentially be performed in parallel.

Flexibility

The ability to treat functions as first-class objects, and the ability to encapsulate an environment (set of variable bindings) into a function in the form of a *closure* can provide a great deal of flexibility in selective collapsing and recombination of protocol layers and functions. The flexibility is reflected in the ease of design and implementation, and also in the modularity that allows operations such as dynamic reconfiguration.

First-class functions also provide the tools to take advantage of the ALF approach. In the absence of first-class functions, it is still necessary for all the sender data to arrive before the receiver application function can be invoked: otherwise, we run the risk of attempting to access data that is not yet present. We address the problem by using a future construct: with first-class functions, the system can be designed to take as arguments functions that, when executed, will produce the relevant arguments. This way, all information about locating and extracting the relevant application

frames remains internal to the appropriate modules, yet the application is free to begin its processing immediately after the receipt of the first application frame. This not only increases flexibility, but has the potential for improving performance as well. As long as the application doesn't actually try to access data that has not yet arrived, it can take advantage of any available concurrency and begin its processing in parallel with any further data transmission. Thus, the processing required on the sender, the transmission itself, and the processing on the receiver can all potentially be performed at least partially in parallel.

Information Flow: Composing Functions

The conventional layered protocol implementations have the advantage from a software engineering perspective of modularity and a clean, well-defined abstraction and interface. The disadvantage is that adherence to the interface specifications results in a narrow, restricted flow of information between the protocol layers. Control flow is restricted to hierarchical calls between adjacent layers, and the data is generally represented as a low-level type such as a string of bytes.

Some of these limitations can be addressed by the inter-layer function composition of ILP. Performing I/O-intensive operations sequentially requires multiple accesses to the same data. Thus, composing the functions and applying them incrementally to the same data has the potential for significant performance improvements. This kind of function composition yields a program structure similar to one designed for lazy evaluation. In both cases, the technique is to store enough state in the closures to rearrange the execution order; in a certain sense, it is irrelevant whether performance or flexibility is the ultimate goal.

Although modularity and efficiency are both desirable properties, applying this mechanism in a conventional programming language environment has the unfortunate effect of putting them in opposition. However, if the appropriate language paradigms are available, we can reorder the processing order of the system by extending the module (layer) interfaces to pass functions rather than values. This leaves the order of processing, an implementation decision, to the implementor, but leaves the specification independent of the implementation details. This is particularly useful for a network protocol specification, since it is imperative that interface specifications be fixed despite the wide variety of local needs and physical configurations.

Streamlining Critical Paths

The protocol processing package provides an example of the use of first-class functions for lazy evaluation.

Choosing alternate evaluation strategies involves making assumptions about the normal behavior of the system. For example, delaying the checksum computation may only be cost-effective if we expect most checksums to succeed, or if we expect some percentage of the transmitted data to be superfluous. This approach requires a fixed overhead proportional to the volume of message data

transmitted in order to enable the sender to resend old packets should a delayed checksum operation fail. The sender must have access to sufficient information to either regenerate the relevant frame, or to extract it from storage. The current implementation stores frame values for a preset timeout period, rejecting any retry requests which arrive after timeout expiration. The length of the timeout puts a bound on the length of time for which the receiver may safely delay checksum evaluation. Cost calculations must consider the resources required to either save or re-compute the values requested by the remote site as well as any performance benefits introduced by the delayed evaluation on the receiver.

In addition, since communication processing is generally a critical path, it is also possible to improve overall performance with strategies such as lazy processing on the receiver (as in ILP/ALF) or eager transmission strategies on the client. Strategies such as lazy, delayed, or eager evaluation (see Section 2) can be readily expressed with various combinations of first class function parameters.

5.4.3 Type System

The advantages and drawbacks of a strong type system are essentially the same for the protocol processing framework as for the previous two case studies. The system implicitly relies on the type system's validation of the types passed across the interfaces, and on the associated guarantees of modularity and locality. In addition, the type system presents the same set of communication problems as discussed previously: the absence of runtime type information, and the problems of accessing abstract type implementation structure.

In the context of ALF, we can consider again what it means to share constructed types, both abstract and non-abstract, across address spaces. In fact, even the term "abstract" itself may be misleading: even for complex types with explicit representations at the language level, the physical representation may differ from compiler to compiler or machine to machine. So, in fact, it is still the type *abstraction* that is being shared rather than any concrete representation. In the case of abstract types with private representations, the shared abstraction is at an even higher level. Such a notion of sharing seems a fragile basis for a mechanism as potentially powerful as ALF, which depends on the notion of common application-level types.

In general we side-step a class of the problem by choosing to synchronize on language-level rather than bit-level type representation, expecting the communication system to perform any necessary bit-level transformations as part of the transfer. However, the notion of abstract data-types challenges even that, and demands reconsideration of the meaning of distributed type sharing. It is possible that the problem will not be adequately addressed without distributing the type management of the language itself; in any case, it is clear that this subject requires further investigation.

5.5 Conclusion

Language constructs such as closures and a type-safe parameterized module system can greatly enhance the power and flexibility of large systems. This is particularly true of distributed systems, where standard systems issues are compounded by reliance on remote communication and the existence of additional degrees of freedom; these manifest themselves, for example, as complex failure modes and independently administered system components. Distribution also requires the ability to deal with multiple event orderings, increasing the relevance and utility of alternate evaluation strategies. This case study shows that passing first-class function arguments across layering boundaries preserves the encapsulation of the protocol implementation, but nevertheless allows reordering and composition of operations. Combining this power with the flexibility introduced by application level framing introduces the potential for a wide variety of alternate evaluation strategies suited to different performance and flexibility requirements.

Rather than fully processing each packet as it arrives, the receiver can create a closure to perform each required function, and associate these with the unprocessed message. Returning this closure to the appropriate presentation routines enables it to be applied during the presentation processing even though it may require access to variables internal to, for example, the transport or checksum modules. The layer boundaries are strictly enforced, but the relevant information is still available at execution time because it is encapsulated within the closure environment.

The protocol processing package again demonstrates the value of a parameterized module system. In this case, the focus is mainly on the value of type-safe encapsulation rather than on configurability, and on transparent layering the communication and ALF modules into the overall system. The runtime type problem did not come up specifically here, since it is implicitly encapsulated in the communication layer and the implementation did not attempt to transmit any polymorphic types; however, the lack of runtime type information is a problem for any system which requires linearization of types whose representation is not fixed.

First-class functions provide the basis for the delayed processing approach, allowing the design to take advantage of the application level framing to remove protocol processing from the critical path and potentially introduce parallelism into the system. When combined with the parameterized module system, they provide the means of reorganizing the control flow of the system without sacrificing the encapsulation of the logical layered model.

Chapter 6

Performance

As discussed earlier, the case studies are intended as design validations rather than production-quality systems. Although they have been implemented in SML/NJ, they represent language-independent examples of systems implemented using specific high-level language features. Absolute performance measurements can provide a reference point, but both a compiler and the applications themselves require more development and tuning before they can be meaningfully compared to production systems implemented in more conventional, highly-tuned languages. Since languages such as Modula-3 and Scheme support many of the relevant language mechanisms and still achieve respectable performance, it is likely that many of the performance bottlenecks in SML and other high-level languages are not inherent in the languages themselves; this is further supported by the results of on-going work focused on improving various aspects of SML/NJ performance.

Micro versus Macro Performance In making a global performance analysis, it is important to realize that the number of variables in any real system makes it impossible to be truly objective. For example, one might perform a study that suggests that it is acceptable for a particular language construct to be expensive because it is rarely used, when in reality it may be precisely *because* it is expensive that it is rarely used. Although it is certainly relevant to consider micro performance, especially for critical sections of code, the overall performance of complex, and especially distributed, systems can depend on higher-level abstractions. Distributing a system introduces a whole new set of issues which extract a cost in resources and performance. Communication itself introduces a certain amount of latency, and suddenly the importance of straight-line code performance must be weighed against higher-level design strategies that trade off argument complexity or computation for message frequency. Communication mechanisms such as broadcast and multicast [16, 11] have demonstrated the effectiveness of parallel communication in reducing communication latency, network bandwidth, and sender computation time. Overlapping portions of the network latency with remote processing time can significantly improve overall performance [52] even when physical

multicast isn't available, especially in cases where group sizes are large.

6.1 SML/NJ

Implementation-Specific Efficiency Considerations

Specific design and implementation decisions can have a significant impact on the performance of any computer system. We examine some of these choices for SML/NJ so that we can consider their effects, both local and global, on the performance of our case studies. For a more detailed discussion of the SML compiler, see Appel [3].

The compiler uses *continuation passing style* (CPS) as an intermediate representation, so functions are automatically represented as closures. SML/NJ's CPS-style implementation adds an extra overhead for procedure calls. Since procedure call frames are allocated on the heap rather than on a stack, each one requires a tag word. Although functional programming style typically results in fewer real procedure calls than imperative style, the calls that are made must include the extra tag-word overhead.

The implementation of SML/NJ keeps all storage, even procedure call frames, on the heap. This means that procedure frames are garbage collected in the same way as any other data. This approach introduces some performance tradeoffs: for instance, storing frames on the heap can incur large garbage collection overheads if it is done naively for highly recursive routines. On the other hand, the heap based approach means that no special operations are required for data that outlives its enclosing block.

Another issue for heap-based languages is memory usage. Not only does a complex program require a lot of heap space, but it also requires enough additional space to do a copying garbage collection. Although it is possible to load a machine with enough memory to avoid paging under these conditions it is unfortunately not always practical, and many users are still experiencing performance degradation due to paging.

One SML/NJ implementation consideration particularly relevant to systems programming is that of using tagged integers. Allocating one bit for the tag means that only 31 bits are left for the integer value, so all integer operations must do overflow checking as well as tagging/untagging. This results in roughly 2 to 3 operations to each one in an environment with untagged 32 bit integers. Although the compiler can optimize away a certain percentage of these checks, and although some specific architectures may be able to use traps in place of checks, there will always be some additional overhead for integer arithmetic with this implementation.

Arrays, which are often used in systems programming, are also slow in SML/NJ. Array overhead comes from two main factors: they require bounds checking, and they are composed of mutable datatypes. Fortunately, some of the mutability overhead is avoided for the case of integer arrays: integers, unlike more complex types, do not require a *cons* operation to the internal store list for each assignment.

The current implementation contains no special optimizations for address manipulations. Such optimizations are not particularly hard to add: their absence merely reflects the lack of tuning for systems programming. A related example is the current implementation of the stringcopy operation, one that is used quite heavily in communication systems.

A more complicated problem is the poor data locality exhibited by languages with copying garbage collectors. Because the garbage collector reorders the data during collection, programs tend to generate a greater number of page faults, cache and translation look-aside buffer misses. Although there is no easy fix for this problem, it is a known problem and is currently under consideration.

The garbage collection algorithm is generational copying, and the resulting overhead is roughly 5 percent of computation time [3]. Since most data is short-lived, generational collection reduces the overhead of collection by separating the older, longer-lived data from the newer data. The older data needs to be collected much less often, which means that the garbage collector can focus on a smaller portion of the total heap space and collect a higher percentage of garbage per run.

Although the garbage collector must maintain large amounts of virtual heap space, much of it is unused at any particular time. Since it maintains so much information about the contents of the heap, the garbage collector can make much better-informed paging decisions than the operating system. There are also many tradeoffs to be made in time versus space optimizations. Current work in this area shows promising reductions in paging activity when the garbage collector's knowledge is combined with Mach's external pager support [15], especially in the domain of heap management [47].

Some performance measurements for the SML/NJ compiler are reported in Appel [3].

6.1.1 SML/NJ Baseline Measurements

To better understand the some of the costs of using SML, we measure some of its basic operations. Some of these are compared with their counterparts in C, a language frequently used in constructing distributed systems. We focus on procedure call, memory allocation, and closure construction. It is important to note that direct comparisons of micro-operation times can be misleading since programs written in SML and C are unlikely to use the same combinations of primitive operations. Not only might similar language constructs be compiled into different instruction sequences, but the entire program structure is likely to be quite different as well.

Measurements were taken for two different machine types: a Decstation 5000 with 64M of memory and a sun 4/330 with 24M of memory. The Decstation 5000 is faster than the Sun 4/330 in all of the tests. C tests are compiled with gcc, and versions both with and without optimization are run. SML tests are run using the SML/NJ [4] compiler in interactive mode.

Since the duration of the operations being measured is well below the granularity of the system clocks¹, the numbers presented are the averages of large numbers of iterations. The general testing

¹The clocks on the systems being tested tick at approximately 15 millisecond intervals

strategy is to run multiple trials, where each trial consists of a sufficient number of iterations to make the time per trial relatively insensitive to variations of clock-tick granularity. Since this varies according to the timing of the operation being measured, the number of iterations is specified as a parameter to the main testing routine. The values recorded per trial are the total time for all n iterations, and the average time for a single iteration (total time divided by n). The mean and standard deviation of the totals and the averages is computed for each set of trials. Although the results are not true values and standard deviations, they nevertheless give a reasonable indication of the variability of the results. In most cases, the standard deviation is quite low, usually below one percent of the total.

The measurements use the timer facility within SML, which estimates garbage collection time as well as mutator (system) time. The data reflect only mutator execution times, although we discuss some of the conditions that result in heavy garbage collection activity at relevant points in the following discussion. Most of the garbage collection activity for these tests was due to the loop effect from the large (usually greater than 100,000) number of iterations required. The data has been adjusted to remove the test overhead, which was measured by running a null loop for each test. The numbers presented do not reflect any garbage collection time. The data used in the graphs is presented in tabular form in Appendix B.

Closure Construction SML/NJ implements procedures as closures, with records containing the function's free variables and a pointer to the function body. As this suggests, Figure 6.1 shows that constructing a closure in SML/NJ involves a fixed overhead time for record construction plus variable time proportional to the number of relevant environment variables.

The performance tests are based on a procedure declaration with 11 arguments. Closure construction time is measured in a tail-recursive loop which generates a list of function closures. The closures are created by partially evaluation the pre-defined curried function, which is passed to the test loop as one of the parameters. Figure B.1 shows the test loop measuring the time for a closure with three bound arguments. The parameter *fcn* refers to the 11 argument curried procedure; the result of applying it to the three arguments given is a function which takes the remaining 8 arguments.

Each iteration of the test loop binds different parameter values in order to ensure the creation of a new closure, and returning the generated list prevents the compiler from optimizing out the computation of otherwise unused values. The use of tail recursion also minimizes the amount of state information (i.e. stack frames) required for the computation.

Memory Allocation Memory allocation is optimized in SML/NJ. All memory, including procedure call frames, are allocated from the heap. Memory allocation and deallocation is implicit in SML: heap space is allocated whenever space is needed to store a value, and is deallocated during periodic garbage collections. Since SML/NJ uses a compacting garbage collector, the known free area is always contiguous. The cost of testing for heap overflow is a single instruction per basic

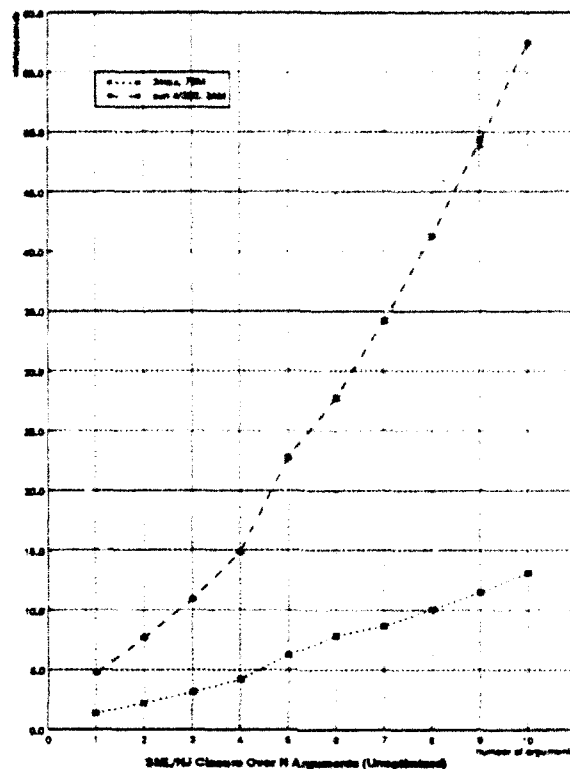


Figure 6.1: Constructing SML/NJ Closures with N Arguments

block. Using the abstract syntax tree of the program, the compiler is able to pre-determine how much storage will be needed by an entire basic block². This is compared against the available free space, and an overflow exception is generated if the available space is insufficient. The exception is caught by the runtime system and causes the invocation of the garbage collector. Allocation therefore requires almost no overhead beyond the data store.

Tests were run for allocation of a tuple of integers (see Figure B.2) ranging in size from 1 to 100. Tuples were used rather than arrays because of the overhead of the continuation construction involved in array creation; the tuple allocation is roughly analogous to a block *malloc* and *bzero* in C. The constructed value in each loop iteration is an *x*-tuple. The results are shown in Figure 6.2.

The C allocation test uses *malloc* to allocate a block of the specified number of (4-byte) words and then initializes it with *bzero* (see Figure B.3). The memory is freed at the end of each trial.

The results are shown in Figure 6.2 and Table B.2. Since the allocation and zeroing are performed

²Dynamically sized objects, such as arrays, are allocated out-of-line, and their allocation code contains explicit overflow checking

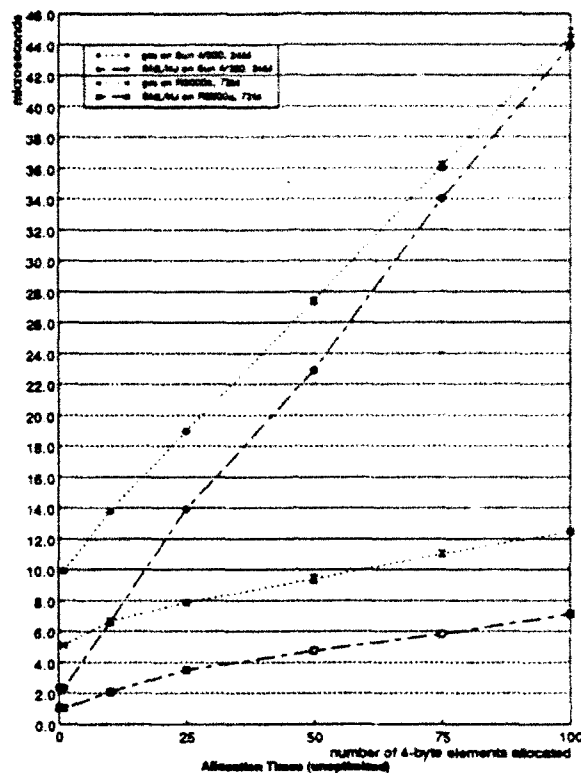


Figure 6.2: Allocation Times

at call time, we expect that the cost of allocation will be greater for a simple block in C than in SML. Optimization had virtually no effect on the performance.

Procedure Call The notion of procedure call in SML, or in functional languages in general, is generally complicated by compiler optimizations based on the language structure. Code written in a functional language is likely to consist of many small, often local procedures. In many cases, the compiler will optimize away the procedural interface and simply in-line the relevant code. One consequence of this is that, despite the appearance of the code, the overall cost for procedure call invocation may often be less for a functional implementation than for a corresponding implementation in an imperative language. Another consequence is that a test routine for measuring procedure call overhead needs to take certain precautions to insure that the procedure call being measured is indeed compiled into a real procedure call.

The procedure call test measures the cost of a procedure call with 0 to 10 arguments. The arguments are touched inside the procedure in order to ensure that they are not optimized away.

Figure B.4 shows the test routine for the 3-argument case. The routine is passed a function of three (integer) arguments, the integers to bind, and the number of iterations. It defines a local function to perform the n iterations of the function invocation, and then invokes the loop. Because the function is defined outside the scope of the test routine and is passed in as a parameter, the compiler cannot know whether or not there are any side effects, and therefore cannot optimize the call away even though no results are returned.

The C procedure call test code (see Figure B.5) is structured similarly to the SML code. The function parameter takes the appropriate number of integer arguments, and adds the sum of their values to a global variable to ensure that the compiler does not optimize out any of the arguments or the function invocation. The results show the same pattern described above, although the performance is significantly better than for the SML/NJ procedure call.

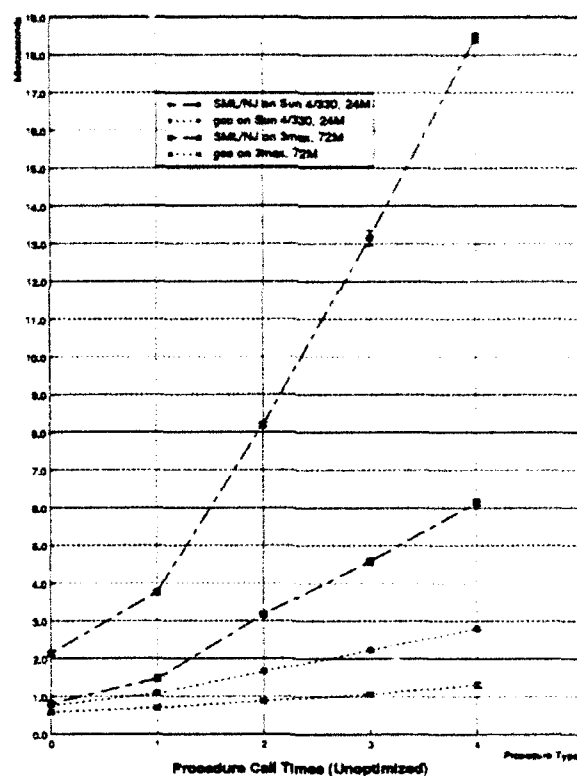


Figure 6.3: Procedure Call Times

The results for the procedure call test are shown in Figure 6.3. Although the optimized and unoptimized versions start off with approximately the same performance for null procedure calls, performance degrades more rapidly at higher numbers of arguments for both machine types.

Chapter 7

Discussion

Distributed systems are more than just an extension of conventional non-distributed systems. They require some method of remote communication, with increasing demand for higher bandwidth and lower latency. Distribution introduces different and often more complex failure modes; this, in turn, requires various combinations of dynamic system reconfiguration, process-independent (persistent) data, more complex system state, and strategies to improve fault-tolerance and overall system performance. System management thus becomes a complicated and error-prone process, and the increased complexity also decreases the validity of intuition in system design, which in turn increases the importance of rapid prototyping as a design tool.

Increasing percentages of programming effort are being devoted to the management of complex distributed state. Like most complex tasks, the work is much easier with the right tools. The language we use shapes our thinking [63], and therefore also shapes the design of our solutions. The flexibility and conceptual simplicity of high-level language abstractions can increase the expressive power and efficiency of the programmer. Studies have shown that the average programmer's productivity, in lines of code, is independent of the programming language used [32]. Since a more expressive language requires fewer lines of code, a programmer using such a language will tend to have a higher productivity than one using a more conventional and less expressive language. Moreover, we are better able to conceive of and express complex systems and protocols with a more powerful and expressive language. Taking advantage of the mechanisms provided by high-level languages can simplify the jobs of the systems designer, the application programmer, and the users. Designing and using programming languages whose features decrease the complexity of the programmer's job while increasing flexibility and safety is an important and perhaps even necessary step to compensate for the growing complexity of distributed systems.

Although distributed systems are more complicated to design and use than their non-distributed counterparts, the realities of scale and current usage pattern seem inescapable. The desire to share data or physical or intellectual resources among physically distributed sites leads immediately to at

least a loosely coupled distributed system. Intellectual resources such as Internet bulletin boards allow reasonably interactive discussions and requests for information among widely physically dispersed individuals with similar interests. Expensive hardware components can be shared among more locally distributed groups. Information such as technical papers or test data can be readily transmitted to connected sites around the world. Although we have moved from time-sharing systems to personal computers or workstations, the data and resource sharing model is still very relevant to our electronic interactions.

Many of the facilities of high-level languages, although not always specifically designed for distributed system or applications, address these concerns. Language supported encapsulation, both at the language and at the module level, provides the implementation independence required to support effective modularity. Sophisticated user-defined types make management of complicated system state easier and more comprehensible, and the built-in modularity of abstract data types make local modifications less disruptive to system availability. Heap based storage makes process-independent data easier to support, and automatic garbage collection avoids much of the risk of unbalanced memory allocation. The flexibility introduced by polymorphism and parameterized module systems facilitates dynamic extensibility and system (re)configuration.

Such structuring mechanisms and techniques are obviously not unique to advanced programming languages, but without the appropriate language support they have hidden pitfalls. Although it is possible to compose crude closure analogs in non-functional programming languages, their complexity makes them awkward to use. The *callbacks* used in the C interface to the X Window System [66] are an example of such awkwardness, as are the call vectors used for runtime branching decisions which are generally composed of globally defined function pointers rather than real closures. These cases require the programmer to ensure that all of the relevant environment is transferred correctly so that it is available at invocation time. Furthermore, the necessity of using a global pointer to define what should be a private (local) function definition means that it is impossible to ensure that an implementation change will not have hidden side effects. The language constructs of an advanced programming language, on the other hand, provide clean, integrated, type-safe abstractions that readily express the modular components of distributed systems and can also provide the basis for more formal reasoning about correctness.

The evidence from the three case studies demonstrates that a functional programming language incorporating support for first-class functions, a powerful type system, and a type-safe parameterized module system can provide an attractive environment in which to implement many types of distributed systems software. The availability of a type-safe parameterized module system allows for straightforward link-time reconfiguration, which in turn provides a basis for simple, rapid prototyping of different design alternatives. It is not necessary to discard the benefits of modularity in order to benefit from layer collapsing, function composition, or delayed evaluation. The availability of first-class functions provides a type-safe way to cross module boundaries without violating the principles of modularity. Furthermore, the ability to dynamically construct closures allows the results of a function application to be as independent as necessary of its invocation environment.

this permits the application of a variety of alternative evaluation strategies to enhance program performance.

In this chapter we discuss the effects of implementing distributed systems software in an environment providing high-level language support. In particular, we focus on issues related to flexibility, control flow, and typing. In each case, we draw on experience and examples from the case study implementations to illustrate the points under consideration.

7.1 Flexibility

Comprehensibility, extensibility, and maintainability are difficult to measure quantitatively but are nontrivial issues as systems become larger and more physically separated. Hardware and software platforms are changing rapidly enough that reliability must include the ability to adapt to the changing environment and changing maintenance crews, features which large software systems often lack. High-level language implementations generally have the advantage of being easier to read and comprehend, since computation constructs are more likely to be described at a high level without a lot of lower-level control information.

Modularity facilitates change by fixing the system interfaces at a more abstract level, leaving more freedom to independently modify the implementation details. A simple possibility is the modification of the implementation of a queue from an array to a list; a more complex possibility is the use of layering to replace a routine body by a communication stub with the same interface.

In addition, a parameterized module system allows for simple mutation of existing interfaces or their specific functionality through the use of wrappers, which provide a functionality loosely analogous to inheritance in an object-oriented language. The exported type of the wrapper is effectively an extension or modification of that of the original module: it takes the functionality of the parameter module as a base and modifies or extends it as desired.

Another manifestation of flexibility is the ability to apply the modularity support to construct higher-level computation structures. This is explored to a small extent with the alternative evaluation strategies proposed in the third case study within the framework of ALF.

Modularity

Modularity is already regarded as a generally advantageous approach to programming in any programming language: one could even argue that it is perhaps the one main precept of software engineering that has been accepted without argument by the systems community. It is clear that even limited application without any special language support can improve the quality and general flexibility of a system.

In fact, modularity can take on many different forms. The most common forms applied in traditional programming languages are groupings of functionality, for example at the level of

procedures or files or subsystems. This sort of modularity is integrated into functional and object-oriented languages, where a more detailed hierarchy of grouping structures is available. In addition, such languages also generally provide *abstraction* or *encapsulation*, providing language support to enforce the privacy (locality) of type, variable, and procedural definitions local to a module. While programs in traditional languages can mimic this general structure, they generally cannot actually enforce any of these locality boundaries.

Another manifestation of language-supported modularity is the first-class function and its implementation as a closure. This extends the encapsulation of data to include procedure bodies (executable code) as well as an associated environment. First-class functions allow procedural abstractions to be passed around in the same way as any other type of data, extending language consistency guarantees to invocation environments.

The modularity embodied in first-class functions is extended further in the form of a parameterized module system. A parameterized module system, once instantiated, binds its module parameters to local names much as a procedure binds its arguments. These parameters may then be referenced throughout the life of the module as a stable part of the module environment. The modularity provided by a parameterized module system can be used to facilitate configurability, by allowing the same module implementation to be instantiated with different parameters just as a procedure can be invoked with different arguments. It can also be used to facilitate extensibility in the form of layering, since module types are defined by their interfaces rather than by their implementations.

	RPC	Linda	Prot
Parameterized Modules	•	•	•
First-class Functions	•	•	•

Figure 7.1: Modularity Support

Figure 7.1 indicates that all three case studies rely on the various forms of modularity. In the ML-RPC case study, we see that the language-supported modularity is used primarily in the form of the parameterized module system, which supports module layering. Although first-class functions can be used to incorporate more complex evaluation strategies, the basic RPC implementation itself does not specifically rely on them. From the ML-Linda example we can see that different configurations of the basic system modules can be selected and specified easily and simply by instantiating the parameterized system components with the appropriate parameters (see Figures 4.1, 4.2, and 4.3). This again relies most heavily on the modularity of the parameterized module system, although as discussed in Chapter 4 the implementation of the eval operator is based on the availability of

integrated first-class functions. Modularity also figures prominently in the third case study, where the ALF implementation allows message frames to be treated as modular units which can be relocated with the help of closures. First-class functions are the most important manifestation of modularity in the ALF implementation, since they are the modular unit used to represent the delayed computations. The third case study also relies heavily on the modularity of the parameterized module system to allow a clean conceptual separation of the different protocol layers.

Although somewhat qualitative, modularity can be evaluated in terms of ease of modification and reconfiguration, as a function of lines of code changed (or added), as a function of time required for the change, and the impact on the overall system (e.g., is the change made at compile-time, link-time, or run-time, and do the changes affect (semi-)independent programs which may interact with the system).

7.2 Control Flow

We also consider the suitability of different control flow mechanisms for different categories of application interfaces. For example, lazy evaluation offers performance enhancements for system interfaces with numerous or complex arguments. Using closures to save the relevant environment for later evaluation reduces the amount of processing that must be done in the critical path; furthermore, it is possible that some arguments or some portions of more complex arguments might not need to be processed at all. A routine that only touches one element of a large array or which only conditionally accesses one or more of its arguments need not take the time to process the unused data.

A great deal of distributed performance enhancement can come out of the right choice of evaluation strategy. It is generally a good strategy to exploit system-specific knowledge when it is available [28, 15, 47]. Consider the analogy of caching to distributed communication: although the simplistic approach is to fetch data on demand, various combinations of pre-fetching and lazy evaluation of large data chunks are generally much more efficient choices.

	RPC	Linda	Prot
Parameterized Modules	•	•	•
First-class Functions	•	•	•
Concurrency	•	•	•
Exceptions	•	•	•

Figure 7.2: Evaluation Strategy Support

Figure 7.2 illustrates the relevance of some specific high-level language features to the support of higher-level evaluation strategies. The ML-RPC system does not take special advantage of evaluation strategies except in that it is structured to allow parallelism in the form of threads. However, its interfaces are designed to take function-valued arguments so that it can support applications using such strategies. The ML-Linda system inherently expects to support at least the degree of parallelism required for the eval operator. The usefulness of lazy evaluation in the implementation of the third case study depends on frame size and the complexity of the data transmitted, as well as the granularity of fragmentation. These data must be considered in conjunction with the relative frequency of retry requests (e.g., dropped packets or checksum failures), since a retry in the presence of delayed evaluation is likely to result in additional processing at the sender.

7.3 Type Issues

Type issues for distributed systems can be broken down into three basic categories. The first includes the safety and verification issues related to type-checking, both locally and across address space boundaries. The second includes the issues for a typed language related to the availability of run-time type information. Finally, the third includes the set of issues concerning the behavior and transmission of abstract types.

Type Checking

Although we often associate the costs of complex type management with high-level languages, in fact the overhead exists no matter what language is used. High-level language support makes complex types more natural and easier to use because the type management support and its associated overheads are integrated into the language itself. In addition, strong static type checking has the advantage of minimizing runtime overhead at the cost of a slightly longer compilation cycle. This is a worthwhile trade-off for systems that rely on the use of complex types.

In a more conventional language, management of the same complex types needs to be provided by the programmer. The costs are spread out over program development, additional management code, and debugging and maintenance time. Thus some of the associated overheads are avoided not by any inherent language efficiency, but by the tendency of the programmer to avoid the difficulties of providing reliable type management for each new complex type. The advantages of the language support for type-checking and management are visible in the structure and flexibility of all of the case studies; they are especially evident during the actual development process.

An issue specific to distributed systems is type checking of messages which cross environment instantiation boundaries. Such communication is explicitly outside the range of language type-checking, and potentially compromises the correctness of an entire program. The encoding and decoding routines of many communication systems have some form of high-level type checking,

but a certain level of confidence is required to translate an incoming stream of bits into a language type. Although a communication system implemented in a strongly-typed high level language can end up with meaningless or garbled input data, it cannot end up with data that will corrupt an entire system or address space. In addition, self-describing encoding schemes can increase the confidence in the validity of the incoming data as the descriptors conform to expected inputs. Nevertheless, absolute safety is impossible as long as the communicating entities are operating independently.

Runtime Type Information

From a language implementor's point of view, a properly designed language with strong static type-checking should not need any runtime type information. Ensuring the existence of operations with properly typed interfaces is sufficient type information for abstract or polymorphic types as well as for more standard explicit types. Unfortunately, this perspective contains some hidden assumptions about the uniformity of type representations which are not necessarily valid for programs that inter-operate across the boundaries of a single language environment instantiation. In the absence of language mechanisms for explicit distribution, communication systems need to incorporate their own set of mechanisms to ensure, to the best of their ability, type compatibility between communicating peers. Furthermore, the type checking guarantees apply only to the needs of the system itself: removing runtime type information makes no allowances for any application-dependent needs¹.

The issue of runtime type information is the most directly relevant to the domain of the thesis because of its focus on communication. In order to transmit data in a heterogeneous environment, it is necessary either to convert between pairs of mutually known data formats, or to choose a fixed transmission format to be shared by all communicating parties. Either case requires some degree of translation from the language representation to a linearized format suitable for transmission, which necessitates access to the physical structure and/or language-level type information of the data being transmitted. Some languages which rely on static type-checking do not have access to such information at run-time: such languages are not well suited to support remote communication without some modification. We see this problem in all three of the case studies.

For base types or explicitly defined complex types, the runtime type information problem can be addressed with a stub generator: the necessary type information is built into the appropriate communication stub routines rather than being generated dynamically. This approach bends the language typing rules by going outside the language itself (i.e. to the interface specifications) to acquire the type information. Polymorphic types, however, are a special case which require a dynamic solution. In this case, there is no static information to be gathered by the stub generator; only a dynamic mechanism can provide the necessary type information².

¹The Linda system's tuple-matching algorithm is an example of such an application-specific use of runtime type information.

²Once a dynamic mechanism exists, the communication encoding routines could also operate within the language and use the dynamic type information rather than interface specification.

Abstract Types

The addition of abstract types complicates the typing issue. Abstract types are distinct from complex types in that their structure is explicitly hidden within the defining module. The availability of type information is not a problem for normal complex types, since their structure is not hidden. Object oriented languages have an analogous problem, since objects can be considered as abstract data types with an associated set of operations. The main principle of abstract types, that of data hiding, poses a problem in the domain of remote communication. A dynamic typing mechanism would not address the abstract type problem, since from the language point of view the type returned could only be the abstract type itself.

Although the transmission of abstract types constitutes an important step in the application of high-level languages to serious communication systems, it is beyond the scope of this thesis. However, in the context of the case study implementations we briefly discuss some possible ad hoc solutions. The communication-related problems can be addressed statically by bending or breaking the language guarantees of representation-hiding in various ways. Such options all involve a loss of generality at some level, since various parts of the encoding/decoding process become fixed.

One possibility is to allow the stub generator to extract the concrete type representation from the implementation source code and embed the necessary type structure in the communication stub routines. Since the language guarantees the privacy and independence of the concrete type representation, however, there is no language mechanism which can guarantee that a generated stub routine corresponds to the concrete representation of the current type instance. It therefore becomes the responsibility of the user to ensure that the version of the communication stub routine always corresponds to the version of the type implementation from which it was generated.

Another approach is to require that a set of linearize/unlinearize operations be required for every abstract type. By fixing the linearization at type implementation time, this approach limits the choice of external representation format, puts the linearization burden on the programmer, and cannot be verified by the system. A slightly more palatable variation is to require that each abstract data type be extended by system-generated linearization routines: this modification at least removes the linearization burden from the programmer. It still has the drawback of leaving instance matching outside the system, depending on the user or application programmer to provide the correct source for input to linearization routine generation; furthermore, it fixes the encoding rules not only at the time of type implementation but at the time of stub generation or even possibly at stub generator implementation time.

7.4 Future Work

Although the thesis discusses the application of SML to distributed systems, it does not discuss the issues involved in distributing the language itself. SML's static typechecking scheme depends on the uniqueness of its tags to identify types at runtime. Under this scheme, it is impossible

for types declared in two different SML address spaces to be recognized as the same type. In addition, if objects and references to objects may be shared across physically separate SML address spaces, the garbage collection algorithm must be extended to deal effectively with the issues of distribution. Distributed SML introduces a whole new category of issues, but is likely to [strengthen the contributions of SML paradigms to distributed systems].

Since much of the power of a language like SML is in its expressiveness, a communication system should be able to support transmission of any legitimate value. This is being explored for a homogeneous environment[46], but there are many additional issues to be considered in extending this to a heterogeneous environment. There is work being done on various intermediate language representations for remote function transmission and invocation[9]. In particular, the SML/NJ compiler produces intermediate language representations for functions during the course of compilations. It should be possible to save or dynamically re-generate the intermediate representations for remote transmission, and have the final compilation be dynamically invoked on the receiver.

We have done some initial work on identifying the features of SML/NJ that are likely to be heavily used in systems software for distributed systems. Applying that knowledge to the compiler would allow for generated code with significantly improved performance. This type of feedback between systems programmers and language designers and implementors can also be extremely useful in earlier stages of language design and implementation.

Although we have use SML/NJ as the vehicle for our examination of the application of high-level language paradigms to distributed systems, it is by no means the only candidate. As mentioned before, several other languages have various combinations of the relevant language mechanisms. Investigating the differences among the combinations and implementations of these mechanisms in programming languages other than SML/NJ would yield a better understanding of the mechanisms themselves and the implications of different implementation techniques.

A deeper analysis of evaluation alternatives requires a wider range of case studies of systems implementations. For example, one could consider some finer-grained concurrency in network processing by using delayed evaluation (in the concurrent sense) to resolve network destination information while the data is being processed. This approach might be able to make use of multiple processors on small multiprocessor systems.

From a software engineering point of view, a larger scale study might follow the evolution of a modularly designed large system, noting the tendency to reuse component modules, or modify specific system modules rather than redesigning and/or re-implementing the entire system as the needs and expectations of the users change over time.

7.5 Contributions

This dissertation illustrates the benefits and drawbacks of distributed system design and implementation on an advanced programming language platform. It combines the principles of high-level

languages with the performance and functionality required for building distributed systems. Distributed systems software may have much in common with more traditional system software, but the size and scope is much larger and correspondingly more complex to manage and use.

Both high-level programming languages and distributed systems have been the subject of much analysis within their own specific domains, but there has been very little analysis of their interactions. The case study designs and implementations provide a concrete framework and body of data for a preliminary analysis. We see that many of the features of high-level programming languages are well-suited to support the complexities inherent in distributed systems, and there is much to be gained from merging the expertise available in the two areas.

Although the high-level language environment as a whole provides a strong support base for program development and maintenance, the case studies show that most of the advantages specific to distributed systems programming come from a small handful of language mechanisms, primarily a parameterized module system, first-class functions, closures, and runtime type information. Other features clearly have useful and even complementary effects, but in each of the three case studies we see that the same combinations of mechanisms consistently play the dominant role. On the other hand, the same set of mechanisms can be combined in different ways to yield different constructs; in this respect, the data from each of the case studies emphasizes a different application of the same basic mechanisms.

The case studies highlight both some of the advantages and disadvantages in current high-level language implementations in supporting distributed systems programming. The data and corresponding analysis is a starting point for the feedback loop between systems programmers and language developers which is necessary before we can realistically move from experimental to production systems development in such high-level environments.

The case study designs and implementations demonstrate that a range of distributed systems can be expressed naturally and effectively using a set of advanced language mechanisms; in particular, the combination of a parameterized module system, a strong type system, runtime type information, and first-class functions provided the most support. Although no existing advanced languages support this specific subset of language mechanisms, the state of advanced languages has evolved to the point where on-going development is occurring and many of the performance limitations have known, feasible solutions. Thus an integrated set of high-level language mechanisms is an important tool for systems programmers since it is well-suited to address the inherent modularity, complex state and failure modes of distributed systems, and can extend many of the advanced language guarantees to the distributed environment.

7.6 Conclusions

The growing size and complexity of distributed systems transform abstractions such as modularity from stylistic concerns to issues of much more fundamental importance. Modularity, in particular,

is important for distributed systems since distribution is inherently modular: the physical separation of components makes their separate evolution almost inevitable. The ability to reason about and customize system modules in isolation is crucial to effective distributed systems. Furthermore, as programs grow larger, they tend to evolve rather than be replaced. Physical separation and distinct administrative domains also require routine version and environment consistency checks that would have been unnecessary in a strictly local system. Programming language mechanisms which support modular, type-safe interactions greatly increase the safety and coherency of interfaces and implementations by allowing the language to enforce these constraints.

The ability to treat functions as first-class objects and to encapsulate an environment (set of variable bindings) into a function in the form of a *closure* can provide a great deal of flexibility in selective collapsing and recombination of program layers and operations. Passing first-class function arguments across module boundaries preserves the encapsulation of the layer implementation, but nevertheless allows reordering and composition of operations across layer boundaries. This flexibility introduces the potential for a wide variety of alternate evaluation strategies suited to different application requirements. This flexibility is reflected in the ease of design and implementation, and also in the modularity that allows operations such as dynamic reconfiguration.

Even token adherence to modular design is likely to improve programming systems, but the real power comes from the ability to use a set of language constructs in combination within the framework of a language which guarantees the safety of their interactions. Such an advanced programming language can also provide formal semantics and provably correct programming feature interactions, which creates a basis for constructing verification systems for program correctness. Most conventional languages do not enforce interface specifications, nor do they allow private (nested) procedure definitions or offer protection from implicit dependencies on values of global variables. Such interdependencies can be even more complex in a distributed system, where an application programmer may not even have access to all the necessary information about system state: in that case, it is imperative that a systems package be able to reliably encapsulate its private data, so that the integrity both of the system and of the application can be properly maintained.

Bibliography

- [1] Mark B. Abbott and Larry L. Peterson. A language-based approach to protocol implementation. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols*, August 1992. to be published.
- [2] Michael J. Accetta, Robert V. Baron, William Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael W. Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Conference*, pages 93–113, July 1986.
- [3] Andrew W. Appel. *Compiling With Continuations*. Cambridge University Press, 1992.
- [4] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In *Functional Programming Languages and Computer Architecture*, pages 301–324. Springer-Verlag, 1987. Volume 274 of *Lecture Notes in Computer Science*.
- [5] Andrew W. Appel and David B. MacQueen. *Standard ML Reference Manual (Preliminary)*. AT&T Bell Laboratories and Princeton University, September 1990.
- [6] Brian N. Bershad, Dennis T. Ching, Edward D. Lazowska, Jan Sanislo, and Michael Schwartz. A remote procedure call facility for interconnecting heterogeneous computer systems. *IEEE Transactions on Software Engineering*, SE-13(8):880–894, August 1987.
- [7] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure call. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.
- [8] Robert Bjornson, Nicholas Carriero, David Gelernter, and Jerrold Leichter. Linda, the portable parallel. Research Report YALE/DCS/RR-520, Yale University Department of Computer Science, February 1987.
- [9] Guy E. Blelloch and Siddhartha Chatterjee. VCODE: A data-parallel intermediate language. Technical report, Carnegie Mellon University School of Computer Science.

- [10] Nicholas Carriero and David Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323-357, September 1989.
- [11] David R. Cheriton and Willy Zwaenepoel. Distributed process groups in the v kernel. *ACM Transactions on Computer Systems*, 2(2):77-107, May 1985.
- [12] G. Chesson. Protocol engine design. In *Proceedings of the Summer 1987 USENIX Conference*, pages 209-215, June 1987.
- [13] David D. Clark. The structuring of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 171-180. ACM, December 1985.
- [14] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols*, pages 200-208, September 1990.
- [15] Eric Cooper, Scott Nettles, and Indira Subramanian. Improving the performance of SML garbage collection using application-specific virtual memory management. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 43-52, June 1992.
- [16] Eric C. Cooper. *Replicated Distributed Programs*. PhD thesis, University of California Berkeley, 1985.
- [17] Eric C. Cooper and Richard P. Draves. C Threads. Technical Report CMU-CS-88-154, Carnegie Mellon University Computer Science Department, June 1988.
- [18] Eric C. Cooper and J. Gregory Morrisett. Adding Threads to Standard ML. Technical Report CMU-CS-90-186, Carnegie Mellon University School of Computer Science, December 1990.
- [19] Excelan Corporation. *Excelan 1020: Network Interface - User's Guide*, 1985.
- [20] James R. Davin. Alf protocol specification. MIT Laboratory for Computer Science, February 1992.
- [21] Per Gunningberg et al. Application protocols and performance benchmarks. *IEEE Communications Magazine*, 27(6):30-36, 1989.
- [22] Anthony J. Field and Peter G. Harrison. *Functional Programming*. International Computer Science Series. Addison Wesley, 1988.
- [23] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80-112, January 1985.

- [24] David Gelernter. Multiple tuple spaces in Linda. In *Proceedings of the Conference on Parallel Architectures and Languages*, 1989.
- [25] Phillip B. Gibbons. A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Transactions on Software Engineering*, SE-13(1):77-87, 1987.
- [26] Per Gunningberg, Craig Partridge, Teet Sirotkin, and Bjorn Victor. Delayed evaluation of gigabit protocols. In *Proceedings of the Second MultiG Workshop*, June 1991.
- [27] Roger Hayes, Steve W. Manweiler, and Richard D. Schlichting. A simple system for constructing distributed, mixed-language programs. *Software Practice and Experience*, 18(7):641-660, July 1988.
- [28] Maurice Herlihy. Using type information to enhance the availability of partitioned data. Technical Report CMU-CS-85-119, Carnegie Mellon University Department of Computer Science, April 1985.
- [29] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1), February 1986.
- [30] Maurice Herlihy and Barbara Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):528-551, 1982.
- [31] T. Holman and L. Snyder. A transparent coprocessor for interprocessor communication in an MIMD computer. Technical Report TR 87-03-07, Computer Science Department, FR-35, University of Washington, July 1987.
- [32] Frederick P. Brooks Jr. *The Mythical Man-Month: essays on software engineering*. Addison-Wesley, 1975.
- [33] Robert H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4), 1985.
- [34] Hemant Kanakia and David R. Cheriton. The VMP network adapter board (NAB): High-performance network communication for multiprocessors. In *Proceedings of the SIGCOMM 1988 Symposium on Communications Architectures and Protocols*, pages 175-187, August 1988.
- [35] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.
- [36] Frederick Knabe and Linda Leibengood. Architecture issues for an ML IPC interface. CMU SCS Fox Note 2, June 1990.

- [37] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, Massachusetts, 1989.
- [38] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.
- [39] Barbara Liskov, Toby Bloom, David Gifford, Robert Scheifler, and William Weibl. Communication in the mercury system. MIT LCS Programming Methodology Group Memo 59-1, April 1988.
- [40] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of Argus. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, November 1987.
- [41] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381-404, July 1983.
- [42] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *Lisp 1.5 Programmer's Manual*. MIT Press, 2nd edition, 1965.
- [43] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [44] James G. Mitchell, William Maybury, and Richard Sweet. Mesa language manual, version 5.0. Technical Report CSL-79-3, Xerox PARC, April 1979.
- [45] Bruce J. Nelson. *Remote Procedure Call*. PhD thesis, Carnegie Mellon University, 1981. Also published as technical report CMU-CS-81-119.
- [46] Scott Nettles. Support for distributed first-class values in standard ML. Thesis proposal, July 1992.
- [47] Scott Nettles, James O'Toole, David Pierce, and Nicholas Haines. Replication-based incremental copying collection. In *International Workshop on Memory Management*. Springer-Verlag, September 1992. Springer-Verlag Lecture Notes in Computer Science. To appear.
- [48] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.
- [49] Larry Peterson, Norman Hutchinson, Sean O'Malley, and Herman Rao. The x-kernel: A Platform For Accessing Internet Resources. *IEEE Computer*, 23(5), May 1990.

- [50] Chris Reade. *Elements of Functional Programming*. International Computer Science Series. Addison-Wesley, 1989.
- [51] Paul Rovner. Extending Modula-2 to build large, integrated systems. *IEEE Software*, 3(6):46-57, November 1986.
- [52] M. Satyanarayanan and Ellen H. Siegel. Parallel communication in a large distributed environment. *IEEE Transactions on Computers*, 39(3), March 1990.
- [53] Ravi Sethi. *Programming Languages: Concepts and Constructs*. Addison Wesley Series in Computer Science. Addison Wesley, 1989.
- [54] Ellen H. Siegel and Eric C. Cooper. Implementing Linda in standard ML. Technical Report CMU-CS-91-151, Carnegie Mellon University School of Computer Science, October 1991.
- [55] Karen R. Sollins. Copying structured objects in a distributed system. *Computer Networks*, 5:351-359, 1981.
- [56] Karen Rosin Sollins. Private communication, March 1991.
- [57] Andrew S. Tanenbaum. Network protocols. *Computing Surveys*, 13(4):453-489, December 1981.
- [58] Warren Teitelman. The cedar programming environment: A midterm report and examination. Technical Report CSL-83-11, XEROX PARC, June 1994.
- [59] Bernd Teufel. *Organization of Programming Languages*. Springer-Verlag, 1991.
- [60] United States Department of Defense. *Reference Manual for the Ada Programming Language*, February 1983. U.S. Government Printing Office, ANSI/MIL-STD-1815A-1983.
- [61] David A. Watt. *Programming Language Concepts and Paradigms*. International Series in Computer Science. Prentice Hall, 1990.
- [62] Robert A. Whiteside and Jerrold S. Leichter. Using Linda for supercomputing on a local area network. In *Proceedings of the Supercomputing Conference*, pages 192-199, November 1988.
- [63] Benjamin L. Whorf. *Language, Thought, and Reality: Selected Writings*. The M.I.T. Press, 1956.
- [64] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, 3rd edition, 1985.
- [65] The Wollongong Group. *The ISO Development Environment: User's Manual*, November 1989.

- [66] Doug Young. Handling input with the X Toolkit. *Unix World*, 7(2), February 1990.
- [67] Hubert Zimmermann. OSI reference model—The ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, COM-28(4):425–432, April 1980.

Appendix A

SML

This appendix presents a description of the syntax and semantics of SML, and a brief discussion of some of the relevant implementation details. SML is the standardized version of the functional language ML (*meta language*), which was conceived as a medium for finding and performing proofs in a logical language[43]. There are compilers for SML and several related dialects, but any specific references in this thesis will be to the Standard ML of New Jersey (SML/NJ) compiler. For more detailed language descriptions, Milner[43] or Paulson[48]; for implementation details or evaluation, see Appel[3].

A.1 Syntax and Semantics

*Standard ML*¹ is a strongly-typed functional language (with type inference and pattern matching) that supports first-class functions, abstract and polymorphic types, recursion, exception handling, garbage collection, and a powerful module system [43].

Some of the examples will be presented as interpreter sessions. The SML/NJ interpreter prompt is the symbol `'-'`. Programmer input follows the prompt, and is terminated by a semicolon. If the input exceeds a single line, the interpreter prompts with the symbol `'='` after each carriage return. The interpreter then evaluates each semicolon-terminated input, and prints out the a representation of the resulting value on a line without the prompt. If an expression is unnamed, its value is assigned to the special variable *it* so that it can be referenced.

Some code fragments will contain embedded comments, which are delimited by the symbols `'(*'` and `*)'`. The keyword `unit` or the special symbol `()` are used to represent a null value.

¹Strictly speaking, SML is an *impure* functional language because it includes imperative features such as assignment.

Types and Values

All types in SML are immutable except for a few special cases such as arrays and *refs* (reference variables). Variables do not support assignment, but are bound at instantiation to an immutable value which is garbage collected when it is no longer needed.

Types in SML are declared and defined with one of the keywords **type** or **datatype**². The **datatype** keyword, described later, indicates an enumerated type composed of a type name and a set of constructors. A datatype declaration always includes the type representation as well as the type name.

Simple types in SML are declared and defined with the keyword **type** followed by the type name, an equal sign, and the type specification. A simple type can be any base type or any legal combination of base types. SML supports the built-in types **int**, **string**, **array**, **bool**, **real**, **list**, **unit**, and **ref**.

```
type sentence = string
type paragraph = string list
```

Values are declared and defined with the keyword **val**³. Variable declarations consist of the keyword **val** and an identifier followed by a colon and a type specification. Values are defined with **val** followed by the variable name, an optional type declaration, and the variable definition preceded by an equal sign. Since SML uses implicit typing a type declaration is not normally required, although it can be included: it consists of a colon followed by the type name or specification. The following example shows the definitions of two integer values, one explicitly typed and one implicitly typed. In both cases, the form of the interpreter response is the same.

```
- val i1 : int = 3;
  val i1 = 3 : int
- val i2 = 4;
  val i2 = 4 : int
```

Lists are delimited by square brackets (**[]**), and their elements are separated by commas. A list has a specific type, and all elements of a list must be of that type. The empty list is denoted by empty brackets or the reserved keyword **nil**. Lists are always null-terminated.

```
- val ilist = [1, 2, 3];
  val ilist = [1,2,3] : int list
```

Any combination of legal types may be combined into a *tuple*. A tuple declaration is represented by an asterisk-separated list of type identifiers, delimited by parentheses; a tuple value is represented

²**eqtype** is a special case of **type** which is used to indicate abstract types which support equality.

³Note that a value may be a function as well as an object

as a comma-separated list of values, also delimited by parentheses. Tuple components may be identified by their position in the tupled sequence by using the symbol `#` and the integer corresponding to the desired component's position in the tuple sequence, so the order of tuple components is significant.

```
- val t1 = (1, 2);
  val t1 = (1,2) : int * int
- val t2 = (2, 1);
  val t2 = (2,1) : int * int
- t1 = t2;
  val it = false : bool
- #1 t1;
  val it = 1 : int
- #1 t2;
  val it = 2 : int
```

A tuple may be defined and used dynamically, or may be defined and named as a distinct user-defined type. The following example defines the type `imag_num` as a tuple of two integers. The variable `im1`, declared as type `imag_num`, is assigned to the tuple `(1, 4)`. The variable `v` is also assigned to the tuple `(1, 4)`, but without the type declaration. Note that the type of `v` is not `imag_num`, but `int * int`.

```
- type imag_num = int * int;
  type imag_num = int * int
- val im1: imag_num = (1, 4);
  val im1 = (1,4) : imag_num
- val v = (1, 4);
  val v = (1,4) : int * int
```

SML also supports a *record* type. A record is a tuple whose components, or *fields* have labels⁴. Since the fields are explicitly labeled, the order of record components is not significant. A record is enclosed in curly braces (`{...}`); in a record declaration, each field has the form `label : value`, and in a definition or pattern match, each has the form `label = value`. For example, the type `dest` from the PEER signature (see Figure 3.5) is an SML record with two fields: `rpeer` and `rport`.

```
type dest = {rpeer: host, rport: port}
```

If `host1` is a value of type `host`, and `port1` is a value of type `port`, then an object of type `dest` could be created and referenced as follows:

⁴An SML record type is similar to the type usually called *structure* in other programming languages.

```

val dest1 = {rpeer = host1, rport = port1}

- #rport dest1;
val it = port1 : port

```

Sometimes it is useful to define a type that is an abstraction uniting several heterogeneous objects, each of which may in turn be an abstraction. Such types are declared with the keyword **datatype** and a set of *constructors*, each of which represents one of the group of heterogeneous objects. Constructors are separated by the symbol ' | '. Each constructor is effectively an operator, accessible to any scope which can access the datatype definition, which, when applied to some (possibly null) set of types, constructs an instance of the defined type. Consider the following type declaration:

```

datatype dessert = EMPTY | PIE of string | COOKIE of int

```

This defines a new type `dessert`, which can be represented by any one of the constructors above. The constructor `EMPTY` does not take any arguments, but `PIE` takes a string which can describe the type of pie, and `COOKIE` takes an integer which specifies the number of cookies. If none of the constructors took any arguments, then the resulting type would be similar to the enumeration type available in some more traditional languages.

If the datatype definition is included in the signature as well as in the structure, then the type implementation as well as the type is exported to the outside world. If, instead, the signature simply defines the type name, then the internal representation of the datatype is available only within the declaring structure. This is referred to as an *abstract* or *opaque* type.

Just as datatypes can be constructed using their constructors, they can also be analogously deconstructed. SML incorporates a pattern matching facility which can be used to break objects into cases. This facility is used primarily in function definitions, variable definitions, and case statements. A routine which prints out the value of an object of type `dessert` might look as follows:

```

fun print_dessert EMPTY =
    print "dessert is undefined"
| print_dessert PIE p =
    print "dessert is " ^ p ^ " pie"
| print_dessert COOKIE n =
    print "dessert is " ^ makestring n ^
      " cookies"

```

Depending on the case of the function argument, a different function body will be executed. Applying the pattern match identifies the constructor, and binds any constructor arguments to the variable names in the pattern. Once bound, these variables can be used in the function body. The different

function bodies are separated by the symbol `'|'`, just as in the datatype definition. The symbol `'..'` is the SML string concatenation symbol; the function `makestring` is an operation exported by the built-in `Integer` abstraction, and converts an integer value to an equivalent string representation.

Polymorphism is expressed in SML with type variables. SML uses the single-quote character to tag its type variables; the tagged variables are read as the corresponding Greek letters to distinguish them from regular variables. Thus, `'a` is read as *alpha*. For example, the type of a polymorphic array would be `'a array`, where `'a` is the polymorphic variable. The polymorphism is at instance granularity; all elements of any particular array must be of the same type, which is determined by the initialization value.

SML also defines a *reference type*, identified by the keyword `ref`. References are essentially pointers. Refs themselves are immutable; their values are fixed storage locations. Assigning a new value to a `ref` changes the contents of the specified location in storage. `ref` is a polymorphic type used to declare reference types; the same keyword is also a constructor which initializes `ref` values. A `ref` can be updated using the assignment operator `:=`, and its contents can be extracted with the dereferencing operator `!`. The value of `:=` is `unit`, and the value of `!` is the type of the `ref` object being dereferenced.

```
- val uniquifier : int ref = ref 3
  val uniquifier = ref 3 : int ref
- !uniquifier;
  val it = 3 : int
- uniquifier := 6;
  val it = () : unit
- !uniquifier;
  val it = 6 : int
```

Functions

In SML a function-valued object can be defined with the keyword `fn` and the reserved symbol `=>`. For example, the following code defines a function which takes an integer argument and returns an integer result equal to twice the value of the argument:

```
fn i => (2 * i)
```

The above format is generally reserved for unnamed function definitions. Named functions are more conventionally defined with the keyword `fun`:

```
fun double i = (2 * i)
```

Function objects can also be created by applying a defined function to fewer arguments than appear on the left-hand side of its definition. This method is sometimes called *partial application*.

Simple functions take a single argument, which may be either a simple type or a tuple of arbitrary types. A tupled argument is similar to the n procedure call syntax in most conventional programming languages.

```
val f: (int * bool) -> int

fun f (i, b) =
    if b
    then (i + 1)
    else (i - 1)
```

This declaration indicates that `f` is a function which takes a tuple composed of an integer and a boolean and returns an integer. The `->` symbol indicates the return value of a function, so the object `f` is a function object. Its declaration therefore uses the keyword `fun`.

The second way of representing a list of multiple function arguments is by *currying* (see section 2). The result of applying the curried function to its first argument is another function, which can then be applied to the next argument in the list. The declaration of a curried function might look as follows:

```
val greater_than: int -> int -> bool
fun greater_than a b =
    if (a > b)
    then true
    else false
```

Since the function `greater_than` is curried, it can be used to create a new function called `gt5` which takes an integer argument and returns `true` if 5 is greater than its argument, and `false` otherwise. This function can be constructed as follows:

```
val gt5 = greater_than 5
```

Signatures, Structures and Functors

The modular unit in SML is the *structure*; an SML *signature* is an interface specification which enumerates any types, exceptions, variables, constants, and functions exported by an associated structure. A structure must implement all specifications in its signature, although a structure may implement functionality not exported by the signature. This allows the possibility of multiple views of a single structure, where each view exports a subset of the total functionality of the structure. A signature in SML is defined with the keyword `signature` followed by the module name; the signature body is delimited by the keywords `sig` and `end`.

SML structures can be combined hierarchically using *functors*, which are parameterized structures whose parameters are bound at their invocation (link time) to other structure instances of the

```

- signature SIG_1 =
=   sig
=       val a: int
=   end
signature SIG_1 =
    sig
        val a : int
    end

- structure Arg1 : SIG_1 =
=   struct
=       val a = 3
=   end
structure Arg1 : SIG_1

```

Figure A.1: SML Structure Definition

types specified by signature identifiers in the functor header. Each functor can access the exported types, values, and operations of its parameter modules, but only for internal use. The external specification for any SML module (structure or functor) is always statically specified by its signature, and only features specified in the signature are accessible from outside the structure or functor.

A structure is defined as shown in Figure A.1 with the keyword **structure** followed by the module name, and the body is delimited by the keywords **struct** and **end**. A functor is defined similarly, as shown in Figure A.1, although the defining keyword is **functor** rather than **structure**, and the module name is followed by a (possibly empty) list of parameter modules and their type declarations.

In functor **Funct**, the value of **b** is dependent on the implementation of the parameter assigned to **A**. The types are all known at compile time, but the values are bound only when the functor arguments are instantiated:

```

- structure F = Funct (structure A = Arg1);
structure F : SIG_2
- F.b;
val it = 3 : int

```

```

- signature SIG_2 =
=   sig
=     val b: int
=   end
signature SIG_2 =
  sig
    val b : int
  end

- functor Funct (structure A: SIG_1) : SIG_2 =
=   struct
=     val b = A.a
=   end
functor Funct : <sig>

```

Figure A.2: SML Functor Definition

A.2 Implementation

Note that it is inexpensive to pass lists as function parameters because a list is defined by a pointer to its first element[53].

Further details on the syntax, semantics, and implementation(s) of SML can be found in [5, 43, 4, 48, 3]

Appendix B

Data for SML and C Baselines

```
fun loop3 fcn n =  
  let  
    val foo = (fcn n n n)  
  in  
    if (n = 0) then (foo)  
    else foo :: (loop3 fcn (n - 1))  
  end
```

Figure B.1: Curried Function Application

```
(* loop function for 25-tuple *)  
fun loopfn n =  
  let  
    val a1 = (n,n,n,n,n,n,n,n,n,n,  
              n,n,n,n,n,n,n,n,n,n,  
              n,n,n,n,n)  
  in  
    if (n = 0) then [a1]  
    else a1 :: (loopfn (n - 1))  
  end
```

Figure B.2: SML Allocation Test

Free Variables	3max (R3000a, 72M)	Sun4/330
Loop overhead	0.49 (0.01)	1.46 (0.02)
1	1.83 (0.02)	6.18 (0.04)
2	2.69 (0.01)	9.12 (0.08)
3	3.62 (0.02)	12.36 (0.12)
4	4.68 (0.01)	16.28 (0.10)
5	6.72 (0.02)	24.22 (0.19)
6	8.34 (0.02)	29.20 (0.19)
7	9.16 (0.03)	35.65 (0.19)
8	10.55 (0.03)	42.74 (0.16)
9	11.97 (0.04)	50.58 (0.39)
10	13.61 (0.04)	58.88 (0.10)

Table B.1: SML Closure Construction Times (microseconds)

```

void alloctest (int trials, int iters, int elts)
    long *intlist[iters];
    int i = 0, j = 0;
    int sz = sizeof(long) * elts;
    struct timeval t1, t2;
    ...
    /* do the timed alloc test */
    gettimeofday (&t1, 0);
    for (i = 0; i < iters; i++)
        intlist[i] = (long *) malloc (sz);
        bzero ((char *) intlist[i], sz);

    gettimeofday (&t2, 0);
    ...

```

Figure B.3: C Allocation Test

Block Size	3max/SML	3max/gcc	Sun4/SML	Sun4/gcc
1	1.03 (0.08)	5.11 (0.07)	2.33 (0.07)	9.91 (0.03)
10	2.09 (0.11)	6.67 (0.17)	6.60 (0.73)	13.80 (0.08)
25	3.52 (0.17)	7.89 (0.15)	13.92 (1.64)	18.95 (0.04)
50	4.75 (0.20)	9.40 (0.29)	22.91 (2.33)	27.40 (0.21)
75	5.84 (0.33)	11.03 (0.21)	34.00 (4.99)	36.13 (0.28)
100	7.13 (0.32)	12.47 (0.10)	43.95 (4.06)	44.50 (0.53)

Table B.2: Block Allocation Times (microseconds)

```

fun procloop3 fcn a1 a2 a3 n =
  let
    fun aloop 0 = fcn a1 a2 a3
      | aloop i = (fcn a1 a2 a3; aloop (i - 1))
    in
      aloop n
  end

```

Figure B.4: SML Procedure Call Test Loop for 3 Arguments

```

void loopproc3 (int iters, void *fcn(),
               int a1, int a2, int a3)
{
  int i = 0;
  for (i = 0; i < iters; i++)
    sumproc ();
    fcn (a1, a2, a3);
}

```

Figure B.5: C Procedure Call Test Loop for 3 Arguments

Arguments	SML/3max	gcc/3max	SML/Sun4	SML/Sun4
Loop	0.96 (0.02)	1.00 (0.00)	2.26 (0.07)	1.26 (0.01)
0	1.77 (0.01)	1.58 (0.02)	4.40 (0.07)	2.02 (0.02)
1	2.43 (0.01)	1.70 (0.04)	6.02 (0.06)	2.34 (0.05)
2	4.13 (0.02)	1.88 (0.02)	10.45 (0.08)	2.92 (0.01)
3	5.54 (0.02)	2.06 (0.03)	15.41 (0.20)	3.49 (0.02)
4	7.11 (0.03)	2.31 (0.07)	20.71 (0.12)	4.06 (0.03)

Table B.3: Procedure Call Times (microseconds)