DESIGN AND IMPLEMENTATION OF A DISTRIBUTED SYSTEMS LANGUAGE

by

Michael Lee Scott

Computer Sciences Technical Report #596

May 1985

DESIGN AND IMPLEMENTATION OF A DISTRIBUTED SYSTEMS LANGUAGE

by

MICHAEL LEE SCOTT

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

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN — MADISON

1985

ABSTRACT

In a distributed environment, processes interact solely through the exchange of messages. Safe, convenient, and efficient communication is of vital importance, not only for the tightly-coupled components of parallel algorithms, but also for more loosely-coupled users of distributed resources. Server processes in particular must be able to communicate effectively with clients written at widely varying times and displaying largely unpredictable behavior. Such communication requires high-level language support.

Interprocess communication can be supported by augmenting a conventional sequential language with direct calls to operating system primitives, but the result is both cumbersome and dangerous. Convenience and safety are offered by the many distributed languages proposed to date, but in a form too inflexible to support loosely-coupled applications. A new language known as LYNX overcomes the disadvantages of both these previous approaches.

The name of the language is a play on its use of duplex communication links. Links are a mechanism for the naming, protection, and abstraction of distributed resources. They allow the connections between processes to be inspected and altered dynamically. Additional language features support the division of processes into multiple threads of control. The interaction of threads and links facilitates the construction of servers.

Experience with LYNX indicates that the language is a significant improvement over existing notations for interprocess communication. An implementation on top of the Charlotte distributed operating system presented several interesting problems and yielded unexpected insights into the nature of the language/operating system interface. A paper design of an implementation for the SODA distributed operating system was in some ways considerably simpler. The Charlotte implementation is complete and performs well.

ACKNOWLEDGMENTS

Many people can contribute to the success of a document with only one name on the title page. This section serves to acknowledge my debt to all those many others.

The most personal thanks are of course non-technical. The contributions of my parents, Dorothy Scott and Peter Lee Scott, to my education, values, and general well-being are beyond estimation. Equally great is the debt to my dear wife Kelly Flynn, who taught me that the important things in life have nothing to do with computer science.

Credit for much of the work described herein belongs to my tireless advisor, Associate Professor Raphael Finkel. If I ever learn to think off-line as well as Raphael does in real time, I'll be doing very well. Close behind Raphael comes his colleague, Marvin Solomon. As a principal investigator for the Charlotte project and as the teacher of several of my formative courses, Marvin has had a major role in shaping my ideas. Members of my final committee deserve thanks, too, for their patience and constructive criticism: Bart Miller, Udi Manber, Terry Millar, Mary Vernon, and Larry Landweber.

Behind the Charlotte and Crystal projects stands a brave and motley crew. Yeshayahu Artsy and Hung-Yang Chang built the current Charlotte kernel. Cui-qing Yang maintains the servers. Nancy Hall put in long hours on the virtual terminal package and the modula compiler. Tom Virgilio built our communications driver and Bob Gerber built its buddy on the hosts. Prasun Dewan, Aaron Gordon, and Mike Litzkow shared my hapless tour of duty as initial users of a untried operating system. Bill Kalsow and Bryan Rosenburg were the projects' knights-errant, keeping us all on our toes. Bryan also built several of the original servers and Bill saved me countless hours of effort by suggesting that my compiler generate C as an "intermediate language."

Generous financial support for my work came from the Wisconsin Alumni Research Foundation (by way of the UW graduate fellowships office), the General Electric Corporation (through their "forgivable loan" program), the National Science Foundation (grant number MCS-8105904), the Defense Advanced Research Projects Agency (contract number N0014/82/C/2087), and AT&T Bell Laboratories (through their doctoral scholarship program).

Chapter 1 was originally written for an independent study course supervised by Raphael Finkel. Intermediate drafts benefited from the written comments of Marvin Solomon and Prasun Dewan. A version very similar to the one included here was published as UW Technical Report #563.

CONTENTS

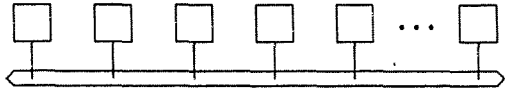
Introduction

The first task of an introduction is to establish definitions. I begin with the words in my title.

I use the adjective **distributed** to describe any hardware or software involving interacting computations on processors that share no physical memory. Distributed algorithms usually entail **concurrency**, that is, they require the simultaneous existence of more than one thread of control. If these threads can execute simultaneously we say they proceed **in parallel**.

The subject area of distributed computing is exceedingly broad. Distributed hardware always consists of **nodes** connected by a **communication medium**, but beyond that very little is fixed. The nodes may be homogeneous or heterogeneous. They may be uniprocessors or multiprocessors. The communication medium can be almost anything, so long as it remains connected. To stay within the realm of feasibility, this dissertation addresses a very narrow subject: a systems programming language for a multicomputer.

As discussed here, a **multicomputer** is a connected network of homogeneous uniprocessors, used as a single machine.



A multicomputer is an attractive hardware option for any organization whose computing load is easily divided into a large number of independent jobs. Interactive timesharing is an obvious example. So long as there are enough jobs to keep its nodes busy, a multicomputer timesharing system offers the advantages

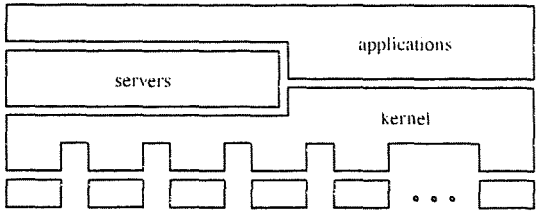
of low response-time variance, graceful degradation in the event of failures, incremental upgrades, and essentially linear gains in throughput with increasing cost.

A multicomputer requires a distributed operating system. Several such operating systems have been built or are under construction [1, 2, 25, 30, 41, 69, 92, 94, 101, 102, 109]. Most employ a relatively small **kernel**, replicated on each node, that cooperates with the hardware to provide the most basic services: communication, low-level device control, and protection. Such traditional operating system functions as resource and device management, routing and directory maintenance, and medium- and long-term scheduling can be provided by **server processes** that run in the same environment as user programs.

There are several reasons for separating servers from the kernel. To the extent that the kernel provides *mechanisms* while the servers set *policy* [120], separation yields the traditional advantages of clarity, ease of maintenance, and the avoidance of mistakes. In addition, considerable amounts of memory can be saved by installing servers on a relatively small number of nodes. Finally, a server responsible for the management of an entire neighborhood of nodes can often make better decisions on the basis of regional information than it could with purely local data.

Together, the kernel and servers constitute the operating system of the multicomputer. They are *systems programs* in the sense that they exist to make the system useful. The kernel runs on a bare machine and implements a new, abstract machine that is safer and easier to use. The servers run on the kernels and tie their machines together. The kernels live in the familiar world of devices

and interrupts on a uniprocessor. They can be written in a conventional systems language. The servers, however, pose new and different problems.



The design of servers is a complicated issue. How many nodes should be covered by a single server? How should the servers in separate neighborhoods interact? How do we balance reliability against redundancy? Such questions are beyond the scope of this dissertation. For my purposes, it suffices to note that the systems programs for a multicomputer will be critically dependent on safe, convenient, efficient, and reliable facilities for interprocess communication. Both servers and utilities (command interpreters, compilers, loaders, and so forth) can be expected to rely on complicated protocols for interprocess communication. Moreover, they must cope with a complicated web of connections to other processes, a web whose topology changes frequently at run time.

One can consider the interconnections among processes on a multicomputer to be a generalization of files. In fact, files themselves may be represented by connections. Where a traditional operating system provides file operations as primitive services, a distributed operating system will provide communication primitives instead. The primitives of existing systems vary quite a bit, particularly with regard to naming, addressing, and error semantics. All, however, allow a user program to request that a message be sent or to wait for a message to

arrive.

It is tempting to suppose that a systems language for a multicomputer could provide communication facilities that translate as directly into operating system primitives as do the file operations of traditional languages. While such a translation might be possible for processes whose communication is limited to file-like operations, it is not possible for processes in general or for servers in particular. The extra complexity of interprocess communication can be attributed to several issues.

(1) Convenience and Safety[1]

Interprocess communication is more structured than are file operations. The remote requests of servers and multi-process user programs resemble procedure calls more than they resemble the transfer of uninterpreted streams of bytes. Processes need to send and receive arbitrary collections of program variables, including those with structured types, without sacrificing type checking and without explicitly packing and unpacking buffers.

(2) Error Handling and Protection[1]

Interprocess communication is more error-prone than are file operations. Both hardware and software may fail. Software is a particular problem, since communicating processes cannot in general trust each other. A traditional file is, at least logically, a passive entity whose behavior is determined by the operations performed on it. A connection to an arbitrary process is much more non-deterministic.

---

[1] Safety involves detecting invalid actions on the part of a single process. Protection means preventing the actions of one process from damaging another.

Fault-tolerant algorithms may allow a server to recover from many kinds of failures. The server must be able to detect those failures at the language level. It must not be vulnerable to erroneous or malicious behavior on the part of clients. Errors in communication with any one particular client must not affect the service provided to others.

(3) Concurrent Conversations

While a conventional sequential program typically has nothing interesting to do while waiting for a file operation to complete, a server usually *does* have other work to do while waiting for communication to complete. Certainly, a server must never be blocked indefinitely while waiting for action on the part of an untrustworthy client. As described by Liskov, Herlihy, and Gilbert [81, 83], and discussed in chapter 3, it is often easiest to structure a server as a dynamic set of tasks, one for each uncompleted request. Efficiency constraints preclude scheduling these tasks in the kernel. Unfortunately, a straightforward translation of the communication primitives provided by most operating systems will include operations that block the calling process, in this case the entire server.

Practical experience testifies to the importance of these issues. The Charlotte distributed operating system [7, 41] is a case in point. As a member of the Charlotte group I have had the opportunity to study the construction of servers firsthand: a process and memory manager (the *starter*), a command interpreter, a process inter-connector, two kinds of file servers, a name server (the *switchboard*), and a terminal driver. Until recently, all were written in a conventional sequential language [40] peppered with calls to the operating system kernel. As work progressed, serious problems arose. The problems can be attributed to the issues just described.

- Charlotte servers devote a considerable amount of effort to packing and unpacking message buffers. The standard technique uses type casts to overlay a record structure on an array of bytes. Program variables are assigned to or copied from appropriate fields of the record. The code is awkward at best and depends for correctness on programming conventions that are not enforced by the compiler. Errors due to incorrect interpretation of messages have been relatively few, but very hard to find.
- Every Charlotte kernel call returns a status variable whose value indicates whether the requested operation succeeded or failed. Different sorts of failures result in different values. A well-written program must inspect every status variable and be prepared to deal appropriately with every possible value. It is not unusual for 25 or 30% of a carefully-written server to be devoted to error checking and handling.
- Conversations between servers and clients often require a long series of messages. A typical conversation with a file server, for example, begins with a request to open a file, continues with an arbitrary sequence of read, write, and seek requests, and ends with a request to close the file. The flow of control for a single conversation could be described by simple, straight-line code except for the fact that the server cannot afford to wait in the middle of that code for a message to be delivered. The explicit interleaving of separate conversations is very hard to read and understand.

The last problem is probably the most serious. In order to maximize concurrency and protect themselves from recalcitrant clients, Charlotte servers break the code that manages a conversation into many small pieces, separated by requests for communication. The servers invoke the pieces individually so that

conversations interleave. Every Charlotte server shares the following overall structure:

```
begin
   initialize
   loop
      wait for a communication request to complete
      determine the conversation to which it applies
      case request.type of
         A :
            restore state of conversation
            compute
            start new request
            save state
         B :
            ...
      end case
   end loop
end.
```

The flow of control for a typical conversation is hidden by the global loop. Saving and restoring state serves two purposes: it preserves the data structures associated with the conversation and it keeps track of the current point of execution in what would ideally be straight-line code. Both these tasks would be handled implicitly if the conversation were managed by an independent thread of control. Data structures would be placed in local variables and the progress of the conversation would be reflected by its program counter.

The complexity of interprocess communication has motivated the design of a large number of distributed programming languages. Many of these languages are described in chapter 1. Most of the designs are convenient and safe. Their communication statements refer directly to program variables and they insist on type security for messages. Many provide special mechanisms for error handling and recovery. Several allow a process to be subdivided into more than one

thread of control.

Unfortunately, none of the languages surveyed was designed with servers in mind. Most were intended to support communication *within* a single distributed program, not *between* separate programs. The issue of protection is never addressed. The network of interconnections is often statically declared. Moreover, without exception, each language proposal either ignores the question of implementation entirely, or else assumes that everything running on the machine will be written in one common language and that the language implementor will have complete control of that machine down to the hardware level.

For servers, a language must maintain the flexibility of explicit kernel calls while providing extensive features to make those calls safer and more convenient. A language that accomplishes these aims is introduced in chapter 2. Known as LYNX, the language is specifically intended for the loosely-coupled processes supported by the kernel of a distributed operating system. The name of the language is derived from its use of communication channels called **links**.

Links are provided as a built-in data type. A link is used to represent a **resource**. The ends of links can be moved from one process to another. Type security is enforced on a message-by-message basis. Servers are free to rearrange their interconnections in order to meet the needs of a changing user community and in order to control access to the resources they provide. Multiple conversations are supported by integrating the communication facilities with the mechanism for creating new threads of control.

The thesis of this dissertation is two-fold: first, that the LYNX programming language is a significant improvement over existing notations for certain

kinds of distributed computing; and second, that it can be effectively implemented on top of an existing operating system. The first half of the thesis is defended in chapter 3. Example programs demonstrate the use of LYNX for problems not solvable with existing distributed languages. Comparisons to equivalent sequential code with direct calls to operating system primitives show that LYNX is safer, easier to use, and easier to read.

The second claim is defended in chapter 4. Two implementations of LYNX are described, one for Charlotte and one for a system called SODA [69]. The implementation effort encountered several interesting problems and yielded some unexpected insights into the nature of the language/operating system interface. Though the design of LYNX was based largely on the primitives provided by Charlotte, the SODA implementation is in some respects considerably simpler. The SODA implementation exists on paper only; the one for Charlotte is in actual use.

Chapter 1

A Survey of Existing Distributed Languages

1. Introduction

It has been recognized for some time that certain algorithms (operating systems in particular) are most elegantly expressed by **concurrent programs** in which there are several independent and, at least in theory, simultaneously active threads of control. On the assumption that the threads interact by accessing shared data, a whole body of research has evolved around methods for synchronizing that access [19, 20, 28, 35, 36, 52, 56, 57]. Even on a conventional uniprocessor, effective synchronization is crucial in the face of context switches caused by interrupts.

With the development of multicomputers it has become practical to distribute computations across multiple machines. This prospect has lent a new urgency to the study of **distributed programs** — concurrent programs in which separate threads of control may run on separate physical machines. There are two reasons for the urgency:

(1) On a multicomputer, a distributed program may solve a problem substantially faster than could its sequential counterpart.

(2) The systems programs for a multicomputer must by their very nature be distributed.

Unfortunately, there is no general consensus as to what language features are most appropriate for the expression of distributed algorithms. Shared data is no longer the obvious approach, since the underlying hardware supports message

passing instead. The alternatives proposed to date show a remarkable degree of diversity. This survey attempts to deal with that diversity by developing a framework for the study of distributed programming languages. The framework allows existing languages to be compared for semantic (as opposed to purely cosmetic) differences. It also facilitates the exploration of new and genuinely different possibilities.

Section 2 presents the framework. Section 3 uses that framework to describe a number of existing languages. No attempt is made to survey techniques for managing shared data. (Good surveys have appeared elsewhere [6].) The evaluations are intentionally biased towards languages that lend themselves to implementation on top of a distributed operating system, where message passing is the only means of process interaction.

2. The Framework

This section discusses major issues in distributed language design:

- processes and modules
- communication paths and naming
- synchronization
- implicit and explicit message receipt
- message screening and multiple rendezvous
- miscellany: shared data, asynchronous receipt, timeout, reliability

The list is incomplete. The intent is to focus on those issues that have the most profound effects on the flavor of a language or about which there is the most controversy in the current literature.

2.1. Processes and Modules

A **process** is a logical thread of control. It is the working of a processor, the execution of a block of code. A process is described by a state vector that specifies its position in its code, the values of its data, and the status of its interfaces to the rest of the world.

A **module** is a syntactic construct that encapsulates data and procedures. A module is a closed scope. It presents a limited interface to the outside world and hides the details of its internal operation.

In a sense, a module is a logical computer and a process is what that computer does. Several language designers have chosen to associate exactly one process with each module, confusing the difference between the two. It is possible to design languages in which there may be more than one process within a module, or in which a process may travel between modules. Such languages may pretend that the processes within a module execute concurrently, or they may acknowledge that the processes take turns. In the latter case the language semantics must specify the circumstances under which execution switches from one process to another. In the former case the language must provide some other mechanism for synchronizing access to shared data.

Modules are static objects in that they are defined when a program is written. Some languages permit them to be nested like Algol blocks; others insist they be disjoint. In some cases, it may be possible to create new **instances** of a module at run time. Separate instances have separate sets of data.

Some languages insist that the number of processes in a program be fixed at compile time. Others allow new processes to be created during execution. Some languages insist that a program's processes form a hierarchy. Special

rules may govern the relationships between a process and its descendants. In other languages, all processes are independent equals. A process may be permitted to terminate itself, and perhaps to terminate others as well. It will usually terminate automatically if it reaches the end of its code.

2.2. Communication Paths

The most important questions about a distributed language revolve around the facilities it provides for exchanging messages. For want of a better term, I define a **communication path** to be something with one end into which senders may insert messages and another end from which receivers may extract them. This definition is intentionally vague. It is meant to encompass a wide variety of language designs.

Communication paths establish an equivalence relation on messages. Senders assign messages to classes by **naming** particular paths (see section 2.3). Receivers accept messages according to class by **selecting** particular paths (see section 2.6.1). Messages sent on a common path enjoy a special relationship. Most languages insert them in a queue and guarantee receipt in the order they were sent. Some languages allow the queue to be reordered.

One important question is most easily explored in terms of the abstract notion of paths: how many processes may be attached to each end? There are four principal options:[2]

[2] These four options correspond, respectively, to the distributed operating system concepts of input ports, output ports, free ports, and bound ports. I have avoided this nomenclature because of the conflicting uses of the word "port" by various language designs.

(1) Many Senders, One Receiver

This is by far the most common approach. It mirrors the client/server relationship found in many useful algorithms: a server (receiver) is willing to handle requests from any client (sender). A single server caters to a whole community of clients. Of course, a server may provide more than one service; it may be on the receiving end of more than one path. Separate paths into a receiver are commonly called **entry points**. In theory, one could get by with a single entry point per server. The advantage of multiple entries is that they facilitate message screening (see section 2.6.1) and allow for strict type checking on each of several different message formats. From an implementor's point of view, multiple entry points into a single receiver are handled in much the same way as multiple senders on a single communication path.

(2) One Sender, Many Receivers

This approach is symmetric to that in (1). It is seldom used, however, because it does not reflect the structure of common algorithms.

(3) Many Senders, Many Receivers

This is the most general approach. In its purest form it is very difficult to implement. The problem has to do with the maintenance of bookkeeping information for the path. In the one-receiver approach, information is conveniently stored at the receiving end. In the one-sender approach, it is kept at the sending end. With more than one process at each end of the path, there is no obvious location. If all information about the status of the path is stored on a single processor, then all messages will end up going through that intermediary, doubling the total message traffic. If the information is distributed instead, there will be situations in which either a) a sender must

(at least implicitly) query all possible receivers to see if they want its message, or b) a receiver must query all possible senders to see if they have messages to send.

Neither option is particularly desirable. Protocols exist whose communication requirements are linear in the number of possible pairs of processes [12, 26], but this is generally too costly. One way out is to restrict the model by insisting that multiple processes on one end of a path reside on a single physical machine. This approach is taken by several languages: messages are sent to *modules*, not processes, and any process within the module may handle a message when it arrives.

(4) One Sender, One Receiver

This approach is the easiest to implement, but is acceptable only in a language that allows programmers to refer conveniently to arbitrary sets of paths. In effect, such a language allows the programmer to "tie" a number of paths together, imitating one of the approaches above.

The preceding descriptions are based on the assumption that each individual message has exactly one sender and exactly one receiver, no matter how many processes are attached to each end of the communication path. For some applications, it may be desirable to provide a **broadcast** facility that allows a sender to address a message to *all* the receivers on a path, with a single operation. Several modern network architectures support broadcast in hardware [104]. Unfortunately, they do not all guarantee reliability. Broadcast will be complex and slow whenever acknowledgments must be returned by each individual receiver.

Several language and operating system designers have attempted to implement *send* and *receive* as symmetric operations (see in particular sections 3.4 and

4.3.2). Despite their efforts, there remains an inherent *asymmetry* in the sender/receiver relationship: data flows one way and not the other. This asymmetry accounts for the relative uselessness of one-many paths as compared to many-one. It also accounts for the fact that no one even discusses the symmetric opposite of broadcast: a mechanism in which a receiver accepts identical copies of a message from all the senders on a path at once.

2.3. Naming

In order to communicate, processes need to be able to **name** each other, or at least to name the communication paths that connect them. Names may be established at compile time, or it may be necessary to create them dynamically. Naming is closely related to processes, modules, and communication paths. Several comments should be made:

- In the typical case of many senders/one receiver, it is common for the sender to name the receiver explicitly, possibly naming a specific path (entry) into the receiver if there is more than one. Meanwhile the receiver specifies only the entry point. It accepts a message from anyone on the other end of the path.
- Compiled-in names can only distinguish among things that are distinct at compile time. Multiple instantiations of a single block of code will require dynamically-created names.
- In languages where messages are sent to modules, it may be possible for names (of module entry points) to be established at compile time, even when the processes that handle messages sent to the module are dynamically created. Processes within a module may be permitted to communicate

with each other via shared data.

Several naming strategies appropriate for use among independent programs on a distributed operating system are not generally found in programming language proposals. Finkel [43] suggests that processes may refer to each other by capabilities, by reference to the facilities they provide, or by mention of names known to the operating system. The link mechanism described in chapter 2 is a similar approach [100]. It is intended to support communication between processes that are designed, compiled, and loaded at widely disparate times. It allows much later binding than one would usually need for the pieces of a single program.

2.4. Synchronization

Since all interprocess interaction on a multicomputer is achieved by means of messages, it is neither necessary nor even desirable for a language to provide synchronization primitives other than those inherent in the facilities for communication. The whole question of synchronization can be treated as a sub-issue of the semantics of the *send* operation [29, 43, 79]. There are three principal possibilities:[3]

(1) No-Wait Send

In this approach the sender of a message continues execution immediately, even as the message is beginning the journey to wherever it is going. The operating system or run-time support package must buffer messages and

[3] In any particular implementation, the process of sending a message will require a large number of individual steps. Conceivably, the sender could be unblocked after any one of those steps. In terms of programming language semantics, however, the only steps that matter are the ones that are visible to the user-level program.

apply back-pressure against processes that produce messages too quickly. If a communication error occurs (for example, the intended recipient has terminated), it may be difficult to return an error code to the sender, since execution may have proceeded an arbitrary distance beyond the point where the *send* was performed.

(2) Synchronization Send

In this approach the sender of a message waits until that message has been received before continuing execution. Message traffic may increase, since the implementation must return confirmation of receipt to the sender of each message. Overall concurrency may decline. On the other hand, it is easy to return error codes in the event of failed transmission. Furthermore, there is no need for buffering or back-pressure (though messages from separate processes may still need to be queued on each communication path).

(3) Remote-Invocation Send

In this approach the sender of a message waits until it receives an explicit reply from the message's recipient. The name "remote invocation" is meant to suggest an analogy to calling a procedure: the sender transmits a message (*input* parameters) to a remote process that performs some operation and returns a message (*output* parameters) to the sender, who may then continue execution. The period of time during which the sender is suspended is referred to as a **rendezvous**. For applications in which it mirrors the natural structure of the algorithm, remote-invocation send is both clear and efficient. Both the original message and the (non-blocking) reply carry useful information; no unnecessary confirmations are involved. As Liskov [79] points out, however, many useful algorithms cannot be

expressed in a natural way with remote invocation.

The choice of synchronization semantics is one of the principal areas of disagreement among recent language proposals. Section 3 includes examples of all three strategies.

2.5. Implicit and Explicit Message Receipt

Lauer and Needham [75] and Cashin [29] discuss a duality between "message-oriented" and "procedure-oriented" interprocess communication. Rather than semantic duals, I maintain that the two approaches are merely varying syntax for the same underlying functionality. What is at issue is whether message receipt is an *explicit* or an *implicit* operation.

In the former case, an active process may deliberately receive a message, much as it might perform any other operation. In the latter case, a procedure-like body of code is activated automatically by the arrival of an appropriate message. Either approach may be paired with any of the three synchronization methods.

Implicit receipt is most appropriate when the functions of a module are externally driven. An incoming message triggers the creation of a new process to handle the message. After the necessary operations have been performed, the new process dies. Alternatively, one may think of the message as awakening a sleeping process that performs its operations and then goes back to sleep, pending arrival of another message. There may be one such "sleeping process" for each of the module's entry procedures, or it may be more convenient to imagine a single sleeper capable of executing any of the entries. If remote-invocation send is used, it may be intuitive to think of the "soul" of a sender as traveling along

with its message. This soul then animates the receiving block of code, eventually returning to its original location (along with the reply message), and leaving that code as lifeless as before. Each of these options suggests a different implementation.

Implicit receipt is a natural syntax for the client/server model. It is better suited than the explicit approach to situations in which requests may arrive at unpredictable times or in which there is no obvious way to tell when the last message has arrived. Explicit receipt, on the other hand, is more appropriate for situations that lack the client/server asymmetry. It is useful for expressing communication among active, cooperating peers, where both parties have useful work to do between interactions. An obvious example is a producer/consumer pair in which both the creation of new data and the consumption of old are time-consuming operations. (See section 9.1 of chapter 2.)

The choice of syntax for message receipt is a second major area of disagreement among recent language proposals. (Synchronization was the first.) StarMod (section 3.11) and NIL (section 3.16) provide both implicit and explicit receipt. Most languages, however, provide a single option only.

2.6. Details of the *Receive* Operation

As noted above, most languages permit multiple senders but only one receiver on each communication path. In addition, they typically allow a process to be non-deterministic in choosing which entry point to serve next; instead of having to specify a particular path, a receiver is free to accept messages from any of a variety of paths on which they may be present.[4] With remote-invocation

[4] Among the languages discussed in section 3, CSP/80 alone [64] provides a similar degree of flexibility for senders. Though it permits only a single sender

send, a receiver may even accept new messages before replying to old. This section discusses techniques for choosing between available messages and for managing more than one concurrent rendezvous.

2.6.1. Message Screening

Assume for the moment that a process may form the receiving end of several communication paths. Further, assume that each of these paths may carry a variety of messages from a variety of senders. In a completely non-deterministic situation, a receiver might be expected to cope with any message from any process on any path. This burden is usually unacceptable. A process needs to be able to exercise control over the sorts of messages it is willing to accept at any particular time. It needs to qualify its non-deterministic options with **guards** that specify which options are open and which are currently closed.

Semantics

There is a wide range of options for message screening semantics. Every language provides some means of deciding which message should be received next. The fundamental question is: what factors may be considered in reaching the decision? The simplest approach is to "hard-code" a list of open paths. In effect, this approach allows the decision to be made at compile time. Most languages, however, allow at least part of the decision to be made at run time. Usually, the programmer will specify a Boolean condition that must evaluate to "true" before a particular message will be accepted. The question now

and receiver on each communication path, the language allows both senders and receivers to choose among several alternative paths, depending on whether anyone is listening on the other end. This added flexibility entails implementation problems similar to those discussed in section 2.2 (3). For a more complete discussion of CSP, see section 3.4.

becomes: on what may the condition depend? It is not difficult to implement guards involving only the local variables of the receiver. Complications arise when a process tries to base its choice on the contents of the incoming messages. In most languages, messages arriving on a particular communication path are ordered by a queue. In a few cases, it may be possible to reorder the queues. In any case, a simple implementation is still possible if path selection or queue ordering depends on some particular well-known **slot** of the incoming message. PLITS and ZENO for example, allow a process to screen messages by sender name (path) and **transaction** slot (see section 3.7).

In the most general case, a language may permit a receiver to insist on predicates involving arbitrary fields of an incoming message. The implementation has no choice but to go ahead and receive a message sight unseen, then look at its contents to see if it really should have done so. Unless unwanted messages can be returned to their sender, the receiver may require an arbitrary amount of buffer space.

Syntax of Guards

The precise way in which guards are specified depends largely on the choice between implicit and explicit message receipt. With implicit receipt, there are two basic options:

(1) The language may allow the execution of an entry procedure to be suspended until an arbitrary Boolean expression becomes true.

(2) The language may allow the procedure to be suspended on a **condition queue** or **semaphore**, with the assumption that action in some other procedure will release it when it is safe to continue.

The first approach is the more general of the two. The second is easier to implement and is generally more efficient. Brinch Hansen discusses the tradeoffs involved ([23], pp. 15-21). Both approaches assume that execution of an entry procedure can be suspended *after* examining an incoming message. Since messages will differ from one instance of the procedure to the next, separate activation records will be required for each suspended entry. Campbell and Habermann [28] suggest the simpler (and more restrictive) approach of allowing guards to involve local data only, and of insisting they occur at the very beginning of their entry procedures. A language that took such an approach would be able to avoid the separate activation records. It would also be less expressive.

Guards are more straightforward with explicit receipt. The most common approach looks something like a Pascal *case* statement, with separate clauses for each possible communication path. Each clause may be preceded by a guard. The physical separation of clauses allows messages of different types to be received into different local variables. In a language with looser message typing (for example PLITS and ZENO, of section 3.7), there may be a statement that specifies receipt into a single variable from any of a set of open paths. An ordinary sequential *case* statement then branches on some field of the message just received.

2.6.2. Multiple Rendezvous

In a language using remote-invocation send, it is often useful for a receiver to be in rendezvous with more than one sender at a time. One ingenious application involves a process scheduler [22, 87]. The scheduler has two entry points: *schedule_me* and *I'm_done*. Every process with work to do calls *schedule_me*. The scheduler remains in rendezvous with all of these callers but one. While

that caller works, the scheduler figures out which process **P** has the next-highest priority. When the worker calls *I'm_done*, the scheduler ends its rendezvous with **P**.

In a language with both remote-invocation send and implicit message receipt, a *module* may be in rendezvous with several senders at one time. If each entry procedure runs until it blocks, then the module is a monitor [57]. If the implementation time-slices among entries, or if it employs a multiprocessor with common store, then the language must provide additional mechanisms for controlling access to the module's common data.

Multiple rendezvous is also possible with explicit message receipt. Several languages require the receive and reply statements to be paired syntactically, but allow the pairs to nest. In such languages the senders in rendezvous with a single receiver must be released in LIFO order. If senders are to be released in arbitrary order, then the reply (or *disconnect*) statement must be able to specify which rendezvous to end. Mutual exclusion among the senders is not an issue, since only one process is involved on the receiving end. Mao and Yeh [87] note that careful location of a disconnect statement can minimize the amount of time a sending process waits, leading to higher concurrency and better performance. Similar tuning is not generally possible with implicit receipt; senders are released implicitly at the end of entry procedures. It would be possible to provide an explicit *disconnect* with implicit receipt (I do so in chapter 2), but it would tend to violate the analogy to sequential procedure calls.

2.7. Side Issues

The issues discussed in this section are less fundamental than those addressed above. They fall into the category of convenient "extra features"—

things that may or may not be added to a language after the basic core has been designed.

(1) Shared Data

In order to permit reasonable implementations on a multicomputer, a distributed language must in general insist that interaction among processes be achieved by means of messages. For the sake of efficiency, however, a language may provide for shared access to common variables by processes guaranteed to reside on the same physical machine. It may be necessary to provide additional machinery (semaphores, monitors, critical regions, etc.) to control "simultaneous" access.

(2) Asynchronous Receipt

Several communication schemes place no bound on the length of time that can pass before a message is noticed at the receiving end of its communication path. There is certainly no such bound for explicit receipt. There are times, however, when it is desirable to receive data as soon as it becomes available. One solution is to equip a receiving module with so-called **immediate** procedures [43] — special entry procedures that guarantee prompt execution. Immediate procedures imply the existence of shared data, since multiple processes may be active in the same module and since execution may switch from one process to another at unpredictable times.

(3) Timeout and Related Issues

In most proposals employing synchronization or remote-invocation send, the sender of a message may be suspended indefinitely if no one is willing to listen to it. Likewise a process that attempts to receive a message may have to wait forever if no one sends it anything. Such delays may be

acceptable in a distributed program where communication patterns are carefully defined and each process is able to assume the correctness of the others. In certain real-time applications, however, and in language systems that attempt to provide for reliability under various sorts of hardware failure, it may be desirable to provide a mechanism whereby a process that waits "too long" times out and is able to take some sort of corrective action.

One particular sort of timeout is especially useful, and may be provided even in cases where the more general facility is not. By specifying a timeout of zero, a process can express its desire to send or receive a message only when such a request can be satisfied immediately, that is when some other process has already expressed its willingness to form the other end of the interaction.

(4) Robustness

When persistent hardware failures are a serious possibility, or when a program is expected to respond in a reasonable fashion to unpredictable real-time events, it may not be possible to hide all errors from the application layer. Programming languages may need to provide special mechanisms for high-level recovery. Liskov's Extended CLU and Argus (section 3.8) are noteworthy examples. The problems involved in providing for reliability in distributed programs have not been adequately investigated. Like many researchers, I ignore them.

(5) Unreliable Send

In certain applications, particularly in the processing of real-time data, speed may be more important than reliability. It may be more appropriate

to send new data than to resend messages that fail. For such applications, a language may provide fast but unreliable messages. Unreliable **broadcast** is particularly interesting, since it can be provided on some architectures at no more cost than point-to-point communication.

3. Several Languages

This section surveys more than two dozen distributed language proposals. For each, it describes how the language fits into the framework of section 2 and then mentions any features that are particularly worthy of note. Languages are considered in approximate order of their publication. For those without the patience of a saint, I particularly recommend the sections on monitor languages, CSP, Distributed Processes, Argus, and Ada.

3.1. Path Expressions

Path Expressions [28, 53] are more of a mechanism than a language. They were invented by Campbell and Habermann in the early 1970's to overcome the disadvantages of semaphores for the protection of shared data. Rather than trust programmers to insert **P** and **V** operations in their code whenever necessary, the designers of path expressions chose to make synchronization rules a part of the declaration of each shared object.

The path expression proposal makes no mention of modules, nor does it say much about the nature of processes. It specifies only that processes run asynchronously, and that they interact solely by invoking the **operations** provided by shared objects. Like the monitors described below, path expressions can be forced into a distributed framework by considering a shared object to be a passive entity that accepts requests and returns replies. Under this model, the proposal

uses remote-invocation send with implicit message receipt. Communication paths are many-one. There may be several identical objects. Processes name both the object and the operation when making a request.

The declaration of a shared object specifies three things: the internal structure of the object, the operations that may be invoked from outside and that are permitted to access the internal structure, and the **path expressions** that govern the synchronization of invocations of those operations. There is no convenient way to specify an operation that works on more than one object at a time.

A path expression describes the set of legal sequences in which an object's operations may be executed. Syntactically, a path expression resembles a regular expression. "**(A, B); {C}; D**", for example, is a path expression that permits a single execution of either **A** or **B** (but not both), followed by one or more simultaneous executions of **C**, followed in turn by a single execution of **D**. There is no restriction on which executions may be performed on behalf of which processes. Reference [28] includes a proof that path expressions and semaphores are equally powerful; each can be used to implement the other.

Path expression solutions to such problems as access control for readers and writers [33] can be surprisingly subtle and complex. Robert and Verjus [95] have suggested an alternative syntax. Like Campbell and Habermann, they dislike scattering synchronization rules throughout the rest of the code. They prefer to group the rules together in a **control module** that authorizes the executions of a set of **operations**. Their synchronization rules are predicates on the number of executions of various operations that have been requested, authorized, and/or completed since the module was initialized. Their solutions to popular problems are both straightforward and highly intuitive.

3.2. Monitor Languages

Monitors were suggested by Dijkstra [36], developed by Brinch Hansen [20], and formalized by Hoare [57] in the early 1970s. Like path expressions, monitors were intended to regularize the access to shared data structures by simultaneously active processes. The first languages to incorporate monitors were Concurrent Pascal [21], developed by Brinch Hansen, and SIMONE [67], designed by Hoare and his associates at Queen's University, Belfast. Others include SB-Mod [13], Concurrent SP/k [59, 60], Mesa [74, 89], Extended BCPL [86], Pascal-Plus [112], and Modula [115]. Of the bunch, Concurrent Pascal, Modula, and Mesa have been by far the most influential. SIMONE and C-SP/k are strictly pedagogical languages. Pascal-Plus is a successor to SIMONE. SB-Mod is a dialect of Modula. C-SP/k has been succeeded by a production-quality language called Concurrent Euclid [61].

In all the languages, a **monitor** is a shared object with operations, internal state, and a number of **condition** queues. Only one operation of a given monitor may be active at a given point in time. A process that calls a busy monitor is delayed until the monitor is free. On behalf of its calling process, any operation may suspend itself by *wait*ing on a queue. An operation may also *signal* a queue, in which case one of the waiting processes is resumed, usually the one that waited first. Several languages extend the mechanism by allowing condition queues to be ordered on the basis of **priorities** passed to the *wait* operation. Mesa has an even more elaborate priority scheme for the processes themselves.

Monitors were originally designed for implementation on a conventional uniprocessor. They can, however, be worked into a distributed framework by considering processes as active entities capable of sending messages, and by considering monitors as passive entities capable of receiving messages, handling them, and returning a reply. This model agrees well with the semantics of Concurrent Pascal and SIMONE, where monitors provide the *only* form of shared data. It does not agree as well with other languages, where the use of monitors is optional. Distributed implementations would be complicated considerably by the need to provide for arbitrary data sharing.

Concurrent Pascal, SIMONE, E-BCPL, and C-SP/k have no modules. In the other four languages surveyed here, monitors are a special kind of module. Modules may nest. In Modula and SB-Mod, the number of modules is fixed at compile time. In Pascal-Plus and Mesa, new instances may be created dynamically. Pascal-Plus modules are called **envelopes**. They have an unusually powerful mechanism for initialization and finalization. Modules in SB-Mod are declared in hierarchical levels. Inter-module procedure calls are not permitted from higher to lower levels. SIMONE, C-SP/k, and Pascal-Plus provide built-in mechanisms for simulation and the manipulation of pseudo-time.

Concurrent Pascal and C-SP/k programs contain a fixed number of processes. Neither language allows process declarations to nest, but Concurrent Pascal requires a hierarchical ordering (a DAG) in which each parent process lists explicitly the monitors to which its children are permitted access. In the six other languages, new processes can be created at run time. Process declarations may be nested in Pascal-Plus. The nesting defines an execution order: each parent process starts all its children at once and waits for them to finish before proceeding. In Mesa, process instances are created by *fork*ing procedures. Mesa compounds the problems of shared data by allowing arbitrary variables to be passed to a process by reference. Nothing prevents an inner procedure from passing a local variable and then returning immediately, deallocating the variable

and turning the reference into a dangling pointer.

Under the distributed model described above, monitor languages use remote-invocation send with implicit receipt. Communication paths are many-one. In languages that permit multiple monitors with identical entries (Concurrent Pascal, Pascal-Plus, and Mesa), the sender must name both the monitor and entry. It also names both in SIMONE, but only because the bare entry names are not visible under Pascal rules for lexical scope. In E-BCPL the sender calls the monitor as a procedure, passing it the name of the operation it wishes to invoke.

The precise semantics of mutual exclusion in monitors are the subject of considerable dispute [6, 54, 62, 68, 71, 85, 91, 114]. Hoare's original proposal [57] remains the clearest and most carefully described. It specifies two bookkeeping queues for each monitor: an **entry** queue and an **urgent** queue. When a process executes a signal operation from within a monitor, it waits in the monitor's urgent queue and the first process on the appropriate condition queue obtains control of the monitor. When a process leaves a monitor it unblocks the first process on the urgent queue or, if the urgent queue is empty, it unblocks the first process on the entry queue instead.

These rules have two unfortunate consequences:

(1) A process that calls one monitor from within another and then waits on a condition leaves the outer monitor locked. If the necessary signal operation can only be reached by a similar nested call, then deadlock will result.

(2) Forcing the signaler to release control to some other waiting process may result in a prohibitive number of context switches. It may also lead to situations in which the signaler wakes up to find that its view of the world has been altered unacceptably.

One solution to the first problem is to release the locks on the outer monitors of a nested *wait*. This approach requires a means of restoring the locks when the waiting process is finally resumed. Since other processes may have entered the outer monitors in the intervening time, those locks might not be available. On a uniprocessor, the problem can be solved by requiring all operations of all monitors to exclude one another in time. Outer monitors will thus be empty when an inner process is resumed. Most of the languages mentioned here use global monitor exclusion. The exceptions are Concurrent Pascal, Mesa, and SB-Mod.

Concurrent Pascal and Mesa provide a separate lock for each monitor instance. Nested calls leave the outer monitors locked. SB-Mod provides a lock for each set of monitors whose data are disjoint. There are two forms of inter-monitor calls. One leaves the calling monitor locked, the other leaves it unlocked. Neither affects monitors higher up the chain. A process that returns from a nested monitor call is delayed if the calling monitor is busy.

The second problem above can be addressed in several ways. Modula [116], SB-Mod, E-BCPL, and C-SP/k all reduce the number of context switches by eliminating the urgent queue(s). Careful scheduling of the uniprocessor takes the place of mutual exclusion. In general, process switches occur only at wait and signal operations, and not at module exit.[5] When the current

[5] E-BCPL timeslices among the runnable processes. Clock interrupts are disabled inside monitor routines. SB-Mod reschedules processes in response to hardware interrupts, but the interrupts are masked at all levels below that of the current process. Interrupted processes are resumed when the current process attempts to return to a lower interrupt level.

process signals, execution moves to the first process on the appropriate condition queue. When the current process waits, execution may move to any other process that is not also waiting.[6] A process that would have been on one of Hoare's entry queues may well be allowed to proceed before a process on the corresponding urgent queue.

Signal operations in Concurrent Pascal cause an automatic return from monitor routines. There is thus no need for an urgent queue. To simplify the implementation, Concurrent Pascal allows only one process at a time to wait on a given condition. Mesa relaxes these restrictions by saying that a signal is only a *hint*. The signaler does not relinquish control. Any process suspended on a condition queue must explicitly double-check its surroundings when it wakes up; it may find it cannot proceed after all, and has to wait again. Wettstein [114] notes that if signals are only hints then it is indeed feasible to release exclusion on all the monitors involved in a nested wait (though Mesa does not do so). Before continuing, a signalled process could re-join each of the entry queues, one by one. After regaining the locks it would check the condition again.

Kessels [71] suggests a different approach to the semantics of conditions. If every queue is associated with a pre-declared Boolean expression, then the signal operation can be dispensed with altogether. When a process leaves a monitor, the run-time support package can re-evaluate the Boolean expressions to determine which process to run next.

[6] The next process to run after a *wait* is always the next runnable process on a circular list. All processes stay on the list in Modula, SB-Mod, and E-BCPL. Their order is fixed. Process switches are slowed unnecessarily by the need to skip over waiting processes. Waiters in C-SP/k are removed from the list, eventually to be re-inserted behind their signaler.

SB-Mod expands on Kessel's proposal. The Boolean expressions for condition queues are optional. *Wait* suspends the caller if the expression is false or was not provided. *Send* (*signal*) transfers control to the first process on the queue if the expression is true or was not provided. A new operation called "*mark*" sets a flag in the first process on the queue. When the current process leaves its monitor, the queue is re-examined. If the expression is true or was not provided, then the marked process is moved to the ready queue. No process switch occurs.

Of all the languages surveyed, SIMONE is truest to Hoare. It does not provide separate entry queues for every monitor, but it does provide an urgent *stack*, with processes resumed in LIFO order.

3.3. Extended POP-2

Kahn and MacQueen [66] have implemented a small but elegant language based on a generalization of **coroutines**. Their language has much in common with CSP (section 3.4, below) but was developed independently.

Process declarations in Extended POP-2 look very much like procedures. There are no modules. Processes share no data. They are instantiated with a *cobegin* construct called "*doco*." The *doco* statement uses a series of **channels** to connect input and output **ports** in the newly-created processes.

Once running, processes can communicate by means of *put* and *get* operations on ports. Given the binding to channels achieved by *doco*, communication paths are one-one. *Send* is non-blocking and buffered. *Receive* is explicit, and names a single port. There is no provision for non-deterministic or selective receipt. Processes with a single input and a single output port may be instantiated with a special functional syntax.

3.4. Communicating Sequential Processes

CSP [58] is not a full-scale language. Rather, it is an ingenious proposal by C. A. R. Hoare for the syntactic expression of non-determinism and interprocess communication. CSP/80 [64], Extended CSP [8], occam [88], and a nameless language by Roper and Barter [96] are all attempts to expand Hoare's syntax into a usable language. I will refer to Extended CSP as E-CSP and to Roper and Barter's language as RB-CSP.

Processes are the central entities in CSP. There are no modules. Regular CSP, E-CSP, occam, and RB-CSP all allow new processes to be created at run time with a modified *cobegin* construct. CSP/80 provides for a fixed number of independent processes, statically defined. Subprocesses in E-CSP and RB-CSP are not visible to their parent's peers. Messages from outside are addressed to the parent. The parent redirects them to the appropriate child. To avoid ambiguity, the E-CSP compiler guarantees that no two subprocesses ever communicate with the same outsider. RB-CSP performs the equivalent checks at run time. None of the CSP languages supports recursion.

Disjoint processes in CSP do not share data; all interaction is by means of a generalization of the traditional concepts of **input** and **output**. In regular CSP, and in CSP/80 and occam, the result is equivalent to explicit receipt and synchronization send. E-CSP provides both synchronization and no-wait send. RB-CSP uses only no-wait send.

Communication paths in CSP are one-one; both sender and receiver name the process at the other end. Forcing the receiver to name the sender prevents the modeling of common client/server algorithms. It also precludes the use of libraries. The four implementations mentioned here address the problem in different ways. CSP/80 lets processes send and receive through **ports**. Sender ports and receiver ports are bound together in a special linking stage. Occam processes send and receive messages through **channels**. Any process can use any channel that is visible under the rules of lexical scope. E-CSP and RB-CSP provide **processname** *variables*. An E-CSP receiver still specifies a sender, but the name it uses can be computed at run time. An RB-CSP receiver does not specify the sender at all. It specifies a message **type** and must be willing to receive from any sender with a matching type.

Communication is typeless in regular CSP and in occam. Types are associated with ports in CSP/80. They are associated with individual communication statements in E-CSP. Individual input and output commands match only if their types agree. RB-CSP provides a special type constructor called **message** with named **slots**, much like those of PLITS (section 3.7). A given process need only be aware of the slots it may actually use.

CSP incorporates Dijkstra's non-deterministic guarded commands [37]. A special kind of guard, called an **input guard**, evaluates to true only if a specified input command can proceed immediately. In regular CSP, and in E-CSP and RB-CSP, there is no corresponding *output* guard to test whether a process is waiting to receive. Hoare notes that the lack of output guards makes it impossible to translate certain parallel programs into equivalent, sequential versions. CSP with input guards alone can be implemented by the usual strategy for many-one communication paths (see section 2.2): information is stored at the receiving end. The provision of output guards as well leads to the usual problems of many-many paths. (For a discussion, see the appendix of Mao and Yeh's paper on communication ports [87].) Moreover, as noted by the designers of CSP/80, the indiscriminate use of both types of guards can lead to implementation-dependent

deadlock. Nonetheless, CSP/80 does provide both input and output guards. The linker prevents deadlock by refusing to connect a sender with output guards to a receiver with input guards.

3.5. Distributed Processes

In the design of Distributed Processes [22], Brinch Hansen has unified the concepts of processes and modules and has adapted the monitor concept for use on distributed hardware.

A Distributed Processes program consists of a fixed number of modules residing on separate logical machines. Each module contains a single process. Modules do not nest. Processes communicate by calling entry procedures (called **common procedures**) defined in other modules. Communication is thus by means of implicit receipt and remote-invocation send. Data can be shared between entry procedures, but not across module boundaries.

An entry procedure is free to block itself on an arbitrary Boolean condition. The main body of code for a process may do likewise. Each process alternates between executing its main code and serving external requests. It jumps from one body of code to another only when a blocking statement is encountered. The executions of entry procedures thus exclude each other in time, much as they do in a monitor. Nested calls block the outer modules; a process remains idle while waiting for its remote requests to complete. There is a certain amount of implementation cost in the repeated evaluation of blocking conditions. Brinch Hansen argues that the cost is acceptable, particularly if every module resides on a separate physical machine.

3.6. Gypsy

Gypsy [51] was designed from the start with formal proofs in mind. Programs in Gypsy are meant to be verified routinely, with automatic tools.

Much of Gypsy, including its block structure, is borrowed from Pascal [65]. There is no notion of modules. New processes are started with a *cobegin* construct. The clauses of the *cobegin* are all procedure calls. The procedures execute in parallel. They communicate by means of **buffer** variables, passed to them by reference. Since buffers may be accessible to more than one process, communication paths are many-many. Sharing of anything other than buffers is forbidden. There is no global data, and no objects other than buffers can be passed by reference to more than one process in a *cobegin*.

Buffers are bounded FIFO queues. Semantically, they are defined by **history** sequences that facilitate formal proofs. *Send* and *receive* are buffer operations. *Send* adds an object to a buffer. *Receive* removes an object from a buffer. *Send* blocks if the buffer is full. *Receive* blocks if the buffer is empty. In the nomenclature of section 2, Gypsy uses no-wait send and explicit receipt, with the exception that back-pressure against prolific senders is part of the language definition. Declared buffer lengths allow the synchronization semantics to be independent from implementation details.

A variation of Dijkstra's guarded commands [37] allows a process to execute exactly one of a number of *sends* or *receives*. The *await* statement contains a series of clauses, each of which is guarded by a send or receive command. If none of the commands can be executed immediately, then the await statement blocks until a buffer operation in some other process allows it to proceed. If more than one of the commands can be executed, a candidate is chosen at ran-

dom. There is no general mechanism for guarding clauses with Boolean expressions.

3.7. PLITS and ZENO

PLITS [39] is an acronym for "Programming Language in the Sky," an ambitious attempt at advanced language design. In the area of distributed computing, it envisions a framework in which a computation may involve processes written in *multiple languages*, executing on heterogeneous machines. ZENO [9] is a single language based heavily on the PLITS design. Its syntax is borrowed from Euclid [73].

A ZENO program consists of a collection of modules that may be instantiated to create processes. Processes are assigned names at the time of their creation. They are independent equals. A process dies when it reaches the end of its code. It may die earlier if it wishes, but it cannot be killed from outside. There is no shared data. *Receive* is explicit. *Send* is non-blocking and buffered. There is only one path into each process, but each message includes a special **transaction** slot to help in selective receipt. A sender names the receiver explicitly. The receiver lists the senders and transaction numbers of the messages it is willing to receive. There is no other means of message screening — no other form of guards. As in CSP (section 3.4), forcing receivers to name senders makes it difficult to write servers. A "pending" function allows a process to determine whether messages from a particular sender, about a particular transaction, are waiting to be received.

The most unusual feature of PLITS/ZENO is the structure of its messages. In contrast to most proposals, there is no strong typing of interprocess communication. Messages are constructed much like the property lists of LISP [93].

They consist of name/value pairs. A process is free to examine the message slots that interest it. It is oblivious to the existence of others.

In keeping with its multi-language, multi-hardware approach, PLITS prohibits the transmission of all but simple types. ZENO is more flexible.

Recent extensions to PLITS [38] are designed to simplify the organization of large distributed systems and to increase their reliability. Cooperating processes are tagged as members of a single **activity**. A given process may belong to more than one activity. It enjoys a special relationship with its peers: it may respond automatically to changes in their status. Activities are supported by built-in atomic transactions, much like those of Argus (section 3.8).

3.8. Extended CLU and Argus

Extended CLU [79, 80] is designed to be suitable for use on a long-haul network. It includes extensive features for ensuring reliability in the face of hardware failures, and provides for the transmission of abstract data types between heterogeneous machines [55]. The language makes no assumptions about the integrity of communications or the order in which messages arrive.

The fundamental units of an Extended CLU program are called **guardians**. A guardian is a module; it resides on a single machine. A guardian may contain any number of processes. Guardians do not nest. Processes within the same guardian may share data. They use monitors for synchronization. All interaction among processes in separate guardians is by means of message passing.

Receive is explicit. *Send* is non-blocking and buffered. Each guardian provides **ports** to which its peers may address messages. New instances of a guardian may be created at run time. New port names are created for each instance.

The sender of a message specifies a port by name. It may also provide a **reply** port if it expects to receive a response. The reply port name is really just part of the message, but is marked by special syntax to enhance the readability of programs. Within a guardian, any process may handle the messages off any port; processes are *anonymous* providers of services. A facility is provided for non-deterministic receipt, but there are no guards; a receiver simply lists the acceptable ports. In keeping with the support of reliability in the face of communication failure, a timeout facility is provided.

Argus [82, 84] is the successor to Extended CLU. Argus uses remote-invocation send and implicit message receipt. Instead of ports, Argus guardians provide **handlers** their peers may invoke. Processes are no longer anonymous in the sense they were in Extended CLU. Each invocation of a handler causes the creation of a new process to handle the call. Additional processes may be created within a guardian with a *cobegin*-like construct.

Argus programs achieve robustness in the face of hardware failures with **stable storage** and an elaborate **action** mechanism. Actions are *atomic*; they either **commit** or **abort**. If they commit, all their effects appear to occur instantaneously. If they abort, they have no effect at all. Actions may nest. A remote procedure call is a nested action. Built-in **atomic objects** [110] support low-level actions, and may be used within a guardian to synchronize its processes.

3.9. Communication Port

Like CSP and Distributed Processes, Communication Port [87] is less a full-scale language than a concept on which a language might be based. A Communication Port program consists of a fixed collection of processes. There are no modules. There is no shared data. Processes communicate with remote-invocation send and explicit message receipt.

Each process provides a variety of **ports** to which any other process may send messages. Ports provide strict type checking. Senders name both the receiver and its port. There may thus be several receivers with the same internal structure. The receive statement is non-deterministic. Guards may be placed on its options. The guards must refer to local data only. Receiving a message and returning a reply are independent operations; it is possible for a receiver to be in rendezvous with several senders at one time. The senders may be **released** in any order. Careful placement of release statements is a useful tuning technique that can be used to minimize the length of rendezvous and increase concurrency.

3.10. Edison

Edison [23, 24] is a remarkable language in a number of ways. Based loosely on Pascal, Concurrent Pascal, and Modula, it is a considerably smaller language than any of the three. It seems to be an experiment in minimal language design.

Processes in Edison are created dynamically with *cobegin*. Modules are used for data hiding. Communication is by means of shared data, and mutual exclusion is achieved through critical regions. There are no separate classes of critical regions; the effect is the same as would be achieved by use of a single, system-wide semaphore. Entry to critical regions may be controlled by arbitrary Boolean guards. It is possible to follow a programming strategy in which all shared data is protected by monitors created out of critical regions and modules. It is equally possible to avoid such rules.

Despite its title ("a multiprocessor language"), I question the suitability of Edison for use on multiple processors. The use of critical regions that *all* exclude each other could periodically halt all processors save one. On a multicomputer, shared data is an additional problem. Unless a careful programming style is imposed above and beyond the rules of the language itself, Edison does not fit into the framework of section 2.

3.11. StarMod

StarMod [31, 32] is an extension to Modula that attempts to incorporate some of the novel ideas of Distributed Processes. It provides additional features of its own. Modules and processes are distinct. Modules may nest. There may be arbitrarily many processes within a module. Processes may be created dynamically; they are independent equals. Processes within the same **processor module** may share data. The programmer may influence their relative rates of progress by the assignment of **priorities**.

StarMod provides both explicit and implicit message receipt and both synchronization and remote-invocation send. The four resulting combinations employ a common syntax on the sending end. Communication paths are many-one. A sender names both the receiving module and its entry point. Entries may be called either as procedures or as functions. A *procedural send* allows the sender to continue as soon as its message is received. A *functional send* blocks the sender until its value is returned. Remote-invocation send is thus limited to returning a single value.

On the receiving end, a module may mix its two options, using explicit receipt on some of its communication paths and implicit receipt on the others. The sender has no way of knowing which is employed. A receiver can be changed from one approach to the other without any change to the sender. Libraries can be changed without invalidating the programs that use them. When a message arrives at a implicit entry point, a new process is created to handle the call. When a message arrives at a explicit entry point, it waits until some existing process in the module performs a *receive* on the corresponding **port**. There is no mutual exclusion among processes in a module; they proceed in (simulated) parallel. They may arrange their own synchronization by waiting on semaphores. The explicit *receive* is non-deterministic, but there are no guards on its options. A single receiver can be in rendezvous with more than one sender at a time, but it must release them in LIFO order. Separate calls to the same implicit port will create separate, possibly parallel, processes. Separate processes in a module may receive from the same explicit port.

StarMod was designed for dedicated real-time applications. The StarMod kernel behaves like a miniature operating system, highly efficient and tuned to the needs of a single type of user-level program. Simplicity is gained at the expense of requiring every program to specify the interconnection topology of its network. Direct communication is permitted only between modules that are neighbors in that network. The programmer is thus responsible for routing.

3.12. ITP

The Input Tool Process model [18] is an extension of van den Bos's Input Tool Method [17], an unconventional language for input-driven programs.

An ITP program consists of a collection of processes. There are no modules. Processes do not nest. They share no data. Each process consists of a hierarchical collection of **tools**. A tool looks something like a procedure. It is made available for activation by appearing in the **input rule** of a higher-level

tool. (The root tools are always available.) A tool is actually activated by the completion of lower-level tools appearing in its own input rule. Leaf tools are activated in response to inputs from other processes, or from the user.

Input rules allow message screening. They resemble path expressions (section 3.1). They specify the orders in which lower-level tools may be activated. Unwanted inputs can be disallowed at any layer of the hierarchy.

ITP uses synchronization send with implicit message receipt. Within a process, any tool can send data to any other process. The naming mechanism is extremely flexible. At their most general, the communication paths are many-many. A sender can specify the name of the receiving process, the receiving tool, both, or neither. It can also specify **broadcast** to all the members of a process **set**. A receiver (leaf tool) can accept a message from anyone, or it can specify a particular sender or group of senders. A global communication **arbiter** coordinates the pairing of appropriate senders and receivers.

The current ITP implementation runs on multiple processors, but does not allow the most general many-many communication paths. Syntax for the sequential part of the language is borrowed from C [70].

3.13. Ada

The adoption of Ada [108] by the U. S. Department of Defense is likely to make it the standard against which concurrent languages are compared in future years.

Processes in Ada are known as **tasks**. Tasks may be statically declared. They may also be created at run time. The code associated with a task is a special kind of module. Since modules may nest, it is possible for one task to be

declared inside another. This nesting imposes a strict hierarchical structure on a program's tasks. No task is permitted to leave a lexical scope until all that scope's nested tasks have terminated. A task can be aborted from outside. Tasks may share data. They may also pass messages.

Ada uses remote-invocation send. The sender names both the receiver and its entry point. Dynamically-created tasks are addressed through pointers. Communication paths are many-one. *Receive* is explicit. Guards (depending on both local and global variables) are permitted on each clause. The choice between open clauses is non-deterministic. A receiver may be in rendezvous with more than one sender at a time, but must release them in LIFO order. There is no special mechanism for asynchronous receipt; the same effect may be achieved through the use of shared data. Ada provides sophisticated facilities for timed pauses in execution and for communication timeout. Communication errors raise the TASKING_ERROR exception. A programmer may provide for error recovery by handling this exception.

Since data may be shared at all levels of lexical nesting, it may be necessary for separate tasks to share (logical) activation records. That may be difficult across machine boundaries. More subtle problems arise from the implicit relationships among relatives in the process tree. For example, it is possible for a task to enter a loop in which it repeatedly receives messages until all of its peers have terminated or are in similar loops. The implementation must detect this situation in order to provide for normal termination of all the tasks involved.

3.14. Synchronizing Resources

SR [4, 5] is an attempt to generalize and unify a number of earlier proposals. It appears to have grown out of work on extensions to monitors [3].

An SR program consists of a collection of modules called **resources**. A resource may contain one or more processes, and may export **operations** those processes define. Operations are similar to ports in Extended CLU and entries in Ada. The processes within a resource share data. Neither resources nor processes may nest. There is special syntax for declaring arrays of identical resources, processes, and operations. A procedure is abbreviated syntax for a process that sits in an infinite loop with a receive statement at the top and a *send* at the bottom.

Receive is explicit. Its syntax is based on Dijkstra's guarded commands [37]. Input guards have complete access to the contents of potential messages. Moreover, messages need not be received in the order sent. A receiver may specify that the queue associated with an operation should be ordered on the basis of an arbitrarily complicated formula involving the contents of the messages themselves. It is possible for a process to be in rendezvous with more than one sender at a time. It must release them in LIFO order.

SR provides both no-wait and remote-invocation send. Messages are sent to specific operations of specific resources. Thus each communication path has a single receiving resource and, potentially, multiple senders. Operations can be named explicitly. They can also be referenced through **capability** variables. A capability variable is similar to a record; it consists of several fields, each of which points to an operation of a specific type. Within a resource, a particular operation must be served by only one process.

There are no facilities for asynchronous receipt or timeout. Each operation, however, has an associated function that returns the current length of its queue. This function may be used to simulate a *receive* with timeout zero: the receiver simply checks the queue length before waiting.

3.15. Linda

Linda [47, 48, 49] provides the full generality of many-many communication paths. Processes interact in Linda by inserting and removing **tuples** from a distributed, global **tuple space** (**TS**).[7] Tuple space functions as an associative memory; tuples are accessed by referring to the patterns they contain.

Published papers on Linda do not dwell on the language syntax. It seems to resemble C [70]. Processes are created with a *cobegin*-like construct and can share data in addition to TS. The data can be protected with some sort of mutual exclusion mechanism. There is no mention of modules.

Linda combines no-wait send with explicit message receipt. Tuples are added to TS with the non-blocking *out()* command. They are removed with the *in()* command. A *read()* command (originally called *in*()*) allows tuples to be read without removing them from TS. All three commands take an arbitrary list of arguments. The first is required to be an actual value of type **name**. The rest may be actuals or "formals." An *in()* command succeeds when it finds a tuple in TS that matches all its actuals and provides actuals for all its formals. In *out()* commands, formals serve as "don't care" flags; they match any actual. In *in()* commands, formals are slots for incoming data.

The matching of tuples according to arbitrary patterns of actuals provides a very powerful mechanism for message screening. It also leads to serious implementation problems. Much of the work on Linda involves finding tractable algorithms for managing TS. The language was originally intended for the Stony

[7] also called **structured memory** (**STM**) in early papers.

execute one at a time. New processes are created by a built-in procedure that accepts a procedure name and an array to be used as stack space and returns the **id** of a newly-created process. There is no preemption; a given process continues to run until it explicitly relinquishes control and names the process to be run in its stead.

One goal of Modula-2 is to permit a large variety of process-scheduling strategies to be implemented as library packages. By hiding all coroutine transfers in a library, the programmer can imitate virtually any other concurrent language. The imitations can be straightforward and highly efficient. For a uniprocessor, Modula-2 provides the richness of expression of multiple threads of control at very little cost.

4.2. Nelson's Remote Procedure Call

Nelson's thesis [90] is devoted to the development of a *transparent* mechanism for remote procedure calls. A remote procedure call combines remote-invocation send with implicit message receipt. Transparency is defined to mean that remote and local procedure calls appear to be the same; they share the same

- atomicity semantics,
- naming and configuration,
- type checking,
- parameter passing, and
- exception handling.

Nelson describes a mechanism, called **Emissary**, for implementing remote procedure calls. Emissary attempts to satisfy all five of the "essential properties" listed above, together with one "pleasant property": efficiency. The attempt at transparency is almost entirely successful, and the performance results are quite impressive.

Emissary falls short of true transparency in the area of parameter passing. Not all data types are meaningful when moved to a different address space. Unless one is willing to incur the cost of remote memory accesses, pointers and other machine-specific data cannot be passed to remote procedures. Moreover, *in/out* parameters must be passed by value/result, not by reference. In the presence of aliasing and other side effects, remote procedures cannot behave the same as their local counterparts. So long as programmers insist on pointers and reference parameters, it is unrealistic to propose a *truly* transparent mechanism.

4.3. Distributed Operating Systems

The borderline between programming languages and operating systems is very fuzzy, especially in hypothetical systems. Interprocess communication lies very near the border. It is often difficult to tell whether a particular mechanism is really part of the language or part of the underlying system. Much depends on the degree to which the mechanism is integrated with other language features: type checking, variable names, scope rules, protection, exception handling, concurrency, and so forth. The mechanisms described in this section, at least in their current form, are fairly clearly on the operating system side of the line. This dissertation is a first attempt at incorporating them into the language level.

4.3.1. Links

Links were introduced in the Demos [10] operating system. They have been adopted, in one form or another, by several descendant systems: Arachne (Roscoe) [44, 102], Charlotte [7, 41], and DEMOS/MP [92].

Links are a naming and protection mechanism. In Demos, and in Arachne and DEMOS/MP, a link is a capability to an input port. It connects an arbitrary

Brook microcomputer Network, a wrapped-around grid (torus) architecture.

3.16. NIL

NIL [27, 105] is a language under development at IBM's T. J. Watson Research Center. It is intended for use on a variety of distributed hardware. The current implementation runs on a single IBM 370. Processes are the fundamental program units; there is no separate module concept. There is no shared data; processes communicate only by message passing. The designers of NIL suggest that a compiler might divide a process into parallel pieces if more than one CPU were available to execute it.

Communication paths are many-one. They are created dynamically by connecting **output ports** to an appropriate **input port**. Any process can use the *publish* command to create **capabilities** that point to its input ports. It may then pass the capabilities in messages to other processes that can use them in *connect* commands. All type checking on ports is performed at compile time.

NIL provides both no-wait and remote-invocation send. Remote-invocation sends may be **forwarded**. The process receiving a forwarded message is responsible for releasing the sender. No-wait sends are buffered and *destructive*; variables sent in messages assume an uninitialized "typestate" and can no longer be inspected.

Receive in NIL is explicit. It has two varieties, one to correspond to each type of *send*. **Exceptions** are used to recover from communication errors. There are elaborate rules for propagating exceptions when a process terminates.

4. Related Notions

Each of the proposals described in section 3 has been described in the literature (at least in part) as a high-level language for distributed computing. For one reason or another, the proposals in this section have not. They all contain useful ideas, however, and are worth considering in any discussion of interprocess communication and concurrency.

The survey in section 3 is meant to be reasonably complete. No such claim is made for this section. I have used by own personal tastes in deciding what to include.

4.1. Concurrent Languages

Several early high-level languages, notably Algol-68 [106], PI/I [11], and SIMULA [14], provided some sort of support for concurrent processes, or at least coroutines. These languages relied on shared data for interprocess interaction. They were intended primarily for uniprocessors, and may have been suitable for multiprocessors as well, but they were certainly not designed for implementation on multicomputers. Recently, Modula-2 [117, 118] has re-awakened interest in coroutines as a practical programming tool. In designing Modula-2, Wirth has recognized that even on a uniprocessor, and even in the absence of interrupts, there are still algorithms that are most elegantly expressed as a collection of cooperating threads of control.

Modula-2 is more closely related to Pascal [65] than to the Modula of section 3.2. For the purposes of this survey, the principal difference between the Modulas is that the newer language incorporates a much simpler and more primitive form of concurrency. Processes in Modula-2 are actually **coroutines**; they

number of **holders** to an **owner**. The owner can receive messages from the link. It owns the input port. A holder can send messages to the link. It holds the capability. A holder can create copies of its capability, and can send them in messages on other links. The owner can exercise control over the distribution of capabilities and the rights that they confer.

Where Demos links are many-one, Charlotte links are one-one. Their ends are symmetric. Each process can send and receive. There is no notion of owner and holder. Only one process can access a given end of a given link at a given point in time.

The protection properties of links make them useful for applications that are somewhat loosely coupled — applications in which processes are developed independently and cannot assume that their partners are correct. Typically, a link is used to represent a **resource**. (In a timesharing system, a link might represent a file.) Since a single process may implement a whole collection of resources, and since a single resource may be supported by an arbitrary number of operations, links provide a *granularity* of naming somewhere in between process names and operation names.

4.3.2. SODA

SODA [69] is an acronym for a "Simplified Operating system for Distributed Applications." It might better be described as a communications protocol for use on a broadcast medium with a very large number of heterogeneous nodes.

Each node on a SODA network consists of two processors: a **client processor**, and an associated **kernel processor**. The kernel processors are all alike. They are connected to the network and communicate with their client processors

through shared memory and interrupts. Nodes are expected to be more numerous than processes, so client processors are not multi-programmed.

Communication paths in SODA are many-one, but there is a mechanism by which a process can broadcast a request for server names that *match* a certain pattern. All communication statements are non-blocking. Processes are informed of interesting events by means of software interrupts. Interrupts can be masked.

From the point of view of this survey, the most interesting aspect of the SODA protocol is the way in which it *decouples* control flow and data flow. In all the languages in section 3, message transfers are initiated by the sender. In SODA, the process that initiates an interaction can arrange to send data, receive data, both, or neither. The four options are termed, respectively, **put**, **get**, **exchange**, and **signal**. Synchronization in SODA falls outside the classification system described in section 2.4.

Every interaction between a pair of processes has a **requester** and a **server**. The server feels a software interrupt whenever a requester attempts to initiate a transfer. The interrupt handler is provided with a (short) description of the request. At its convenience, the server can **accept** a request that triggered its handler at some point in the past. When it does so, the transfer actually occurs, and the requester is notified by an interrupt of its own. The programmer is responsible for writing handlers and for keeping track of outstanding requests in both the server and requester. In simple cases, the bookkeeping may be managed by library routines.

5. Conclusion

There is no doubt that the best way to evaluate a language is to use it. A certain amount of armchair philosophizing may be justified (this chapter has certainly done its share!), but the real test of a language is practical experience. It will be some time before most of the languages in section 3 have received enough use to make definitive judgments possible.

One very useful tool would be a representative sample of the world's more difficult distributed problems. To evaluate a language, one could make a very good start by coding up solutions to these problems and comparing the results to those obtained with various other methods. Much of the success of any language will depend on the elegance of its syntax — on whether it is pleasant and natural to use. But even the best of syntax cannot make up for a fundamentally unsound design.

Section 2 has discussed some major open questions. The two most important appear to be the choice of synchronization semantics for the send operation and the choice between implicit and explicit message receipt. I have argued elsewhere [98] that a reasonable language needs to provide a variety of options. Just as a sequential language benefits from the presence of several similar loop constructs, so can a distributed language benefit from the presence of several similar constructs for interprocess communication. It is worth noting that thirty years of effort have failed to produce an ideal sequential language. It is unlikely that the next thirty will see an ideal distributed language, either.

Chapter 2

An Overview of LYNX

1. Introduction

This chapter introduces a new distributed programming language. It provides an overview of concepts discussed in considerably more detail in the following chapter and in the appendix. The language, known as LYNX, was specifically designed for systems programs for a multicomputer. It differs from the languages of chapter 1 in three of the major areas covered by that survey:

Processes and Modules

> Processes and modules in LYNX reflect the structure of a multicomputer. Modules may nest, but only within a machine; no module can cross the boundaries between machines. Each outermost module is inhabited by a single process. Processes share no memory. They are managed by the operating-system kernel and execute in parallel. Multiple threads of control within a process are managed by the language run-time system, but there is no pretense of parallelism among them.

Communication Paths and Naming

> LYNX derives its name from links. Links are pairs of one-one, movable communication paths. The programmer has complete run-time control over the binding of links to processes and names to links. The resulting flexibility allows links to be used for reconfigurable, type-checked connections between very loosely-coupled processes — processes written and loaded at widely disparate times.

Syntax for Message Receipt

Messages in LYNX may be received both explicitly and implicitly. Processes can decide at run time which approach(es) to use when, and on which links.

2. Main Concepts

The three most important concepts in LYNX are the **process**, the **link**, and the **thread of control**. Processes are supported by the operating system. They execute in parallel and interact by exchanging messages on two-way communication links.

Each process begins with a single thread of control, executing the initialization code of its outermost module. It can create new threads itself or can arrange for them to be created automatically in response to incoming messages. Separate threads do *not* execute in parallel; the process continues to execute a single thread until it blocks. It then takes up some other thread where it last left off. If no thread is runnable, the process waits until one is. In a sense, the threads are coroutines, but the details of control transfer are hidden in the run-time support package. Blocking statements are discussed in section 8.

Lexical scope in LYNX is defined as in Modula [115]. New threads of control may created at any level of lexical nesting. Non-global data may therefore be shared by more than one thread. The activation records accessible at any given time will form a tree, with a separate thread corresponding to each leaf. When a thread enters a scope in which a module is declared, it executes the module's initialization code before proceeding. A thread is not allowed to leave a given scope until all its descendants still active in that scope have completed.

The sequential features of LYNX are Algol-like. I will not discuss them here. A full description of the language can be found in the appendix.

3. Links

A link is a two-ended communication channel. Since all data is encapsulated in modules, and since each outermost module corresponds to a single process, it follows that links are the only means of interprocess interaction. The language provides a primitive type called "link." A link variable accesses one end of a link, much as a pointer accesses an object in Pascal [65]. The distinguished value "nolink" is the only link constant.

New values for link variables may be created by calling the built-in function "newlink":

```
endA := newlink ( endB ) ;
```

One end of the new link is returned as the function value; the other is returned through a result parameter. This asymmetry is useful for nesting calls to newlink inside the various communication statements (see below). In practice, calls to newlink seldom appear anywhere else.

Links may be destroyed by calling the built-in procedure "destroy":

```
destroy ( myend ) ;
```

Destroy is similar to "dispose" in Pascal. All link variables accessing *either* end of the link become unusable (i.e. dangling). An attempt to destroy a nil or dangling link is a no-op.

Arbitrary data structures can be sent in messages. If a transmitted data structure contains variables of type *link*, then the link ends referenced by those

variables are moved from the sending process to the receiver. The semantics of this feature are somewhat subtle. Suppose process **A** has a link variable **X** that accesses the "green" end of link **L**. Now suppose **A** sends **X** to process **B**, which receives it into link variable **Y**. Once the transfer has occurred, **Y** will be the *only* variable *anywhere* that accesses the green end of **L**. Loosely speaking, the sender of a link variable loses access to the end of the link involved. This rule ensures that a given end of a given link belongs to only one process at a time.

It is an error to send a link end that is bound to a entry (see below), or on which there are outstanding *sends* or *receives*.

4. Sending Messages

Message transmission looks like a remote invocation:

connect opname (*expr_list* | *var_list*) **on** linkname ;

Run-time support routines package the operation name and expression list into a message and send it out on the link. The current thread in the sender is blocked until it receives a reply message containing values for the variable list.

5. Receiving Messages Explicitly

Any thread of control can receive a message by executing the **accept** and **reply** statements:

accept opname (*var_list*) **on** linkname ;
...
reply (*expr_list*) ;

Accept blocks the thread until a message is available. *Reply* causes the expression

list to be packaged into a second message and returned to the sender. The compiler enforces the pairing of *accepts* and *replies*.

6. Entries

An entry looks much like a procedure. It is used for receiving messages *implicitly*. Entry headers are templates for messages.

entry opname (*in_args*) : *out_types* ;
begin
...
end opname;

All arguments are passed by value. The header may be followed by the keyword *forward* or *remote* instead of a *begin ... end* block. *Remote* has the same meaning as *forward*, except that an eventual appearance of the entry body is not required. Source file inclusion can therefore be used to insert the same entry declarations in both the defining and invoking modules.

Any process may bind its link ends to entries:

bind *link_list* **to** *entry_list* ;

After binding, an incoming request on any of the mentioned link ends may cause the creation of a new thread to execute one of the mentioned entries, with parameters taken from the message. An entry unblocks the sender of the message that created it by executing a reply statement (without a matching *accept*).

A link end may be bound to more than one entry. The bindings need not be created at the same time. A bound end can even be used in subsequent accept statements. These provisions make it possible for separate threads to carry on independent conversations on the same link at more or less the same time. When all of a process's threads are blocked, the run-time support routines

attempt to receive a message on any of the links for which there are outstanding *accepts* or bindings. The **operation name** contained in the message is matched against those of the *accepts* and the bound entries in order to decide which thread to create or resume. If the name differs from those of *all* the outstanding *accepts* and bindings, then the message is discarded and an exception is raised in the sender (see below for a discussion of exceptions).

Bindings may be broken:

unbind *link_list* **from** *entry_list* ;

An attempt to break a non-existent binding is a no-op.

Entries visible under the usual scope rules can be used to create new threads directly, without links or bindings:

call entryname (*expr_list* | *var_list*) ;

The built-in function "curlink" returns a reference to the link on which the request message arrived for the closest lexically-enclosing entry. If there is no enclosing entry, or if the closest enclosing entry was *call*ed locally, then curlink returns nolink. In the examples at the end of this chapter, curlink is used in entries to make and break bindings for the link on which the current request arrived.

In order to facilitate type checking, the operation names and message formats of connect and accept statements must be defined by entry declarations. The entries can of course be declared *remote*.

7. Exceptions

The language incorporates an exception handling mechanism in order to 1) cope with exceptional conditions that arise in the course of message passing, and 2) allow one thread to interrupt another. The mechanism is intended to be as simple as possible. It does not provide the power or generality of Ada [108] or PL/I [11].

Exception handlers may be attached to any *begin ... end* block. Such blocks comprise the bodies of procedures, entries, and modules, and may also be inserted anywhere a statement is allowed. The syntax is

begin
...
when *exception_list* **do**
...
when *exception_list* **do**
...
...
end;

A handler (*when* clause) is executed in place of the portion of its *begin ... end* block that had yet to be executed when the exception occurred.

Built-in exceptions are provided for a number of conditions:

- Failure of the operation name of a message to match an *accept* or binding on the far end of the link.
- Type clash between the sender and receiver of a message.
- Termination of a receiving thread that has not yet replied.
- Destruction of the link.

Links can be destroyed explicitly by threads on either end. They are also destroyed in the event of hardware failures and at process termination.

A built-in exception is raised in the block in which it occurs. If that block has no handler, the exception is raised in the next scope on the dynamic chain. This propagation halts at the scope in which the current thread began. If the exception is not handled at that level, the thread is aborted. If the propagation of an exception escapes the scope of an accept statement, or if an exception is not handled at the outermost scope of an entry that has not yet replied, then an exception is raised in the appropriate thread in the sending process. If the propagation escapes a scope in which nested threads are still active, those threads are aborted recursively.

User-defined exceptions are raised by the statement

raise exception_name ;

A user-defined exception is felt by all and only those threads that have declared a handler for it in some scope on their current dynamic chain (this may or may not include the current thread). Since the handlers refer to it by name, the exception must be declared in a scope visible to all the threads that use it. The coroutine semantics guarantee that threads feel exceptions only when blocked. User-defined exceptions are useful for interrupting a thread that is waiting for something that will never happen.

8. Blocking Statements

As suggested earlier, *connect*, *accept*, and *reply* may cause a context switch by blocking the thread that uses them. A context switch will also occur when control reaches the end of a scope in which nested threads are still active or in which bindings still exist.

There is one additional way to cause a context switch:

await *condition* ;

will guarantee that execution of the current thread will not continue until the (arbitrarily complex) Boolean condition is true.

9. Examples

The sample programs in this section are small and unexciting. They serve as an introduction to the syntax of LYNX.

9.1. Producer and Consumer

The consumer demonstrates explicit receipt of requests. The producer feeds it a continuous stream of data.

```
module producer (consumer : link);

type data = whatever;
entry transfer (info : data); remote;

function produce : data;
begin
  -- whatever
end produce;

begin -- producer
  loop
    connect transfer (produce( )) on consumer;
  end;
end producer.
```

```
module consumer (producer : link);

type data = whatever;
entry transfer (info : data); remote;

procedure consume (info : data);
begin
   -- whatever
end consume;

var buffer : data;

begin -- consumer
   loop
      accept transfer (buffer) on producer; reply;
      consume (buffer);
   end;
end consumer.
```

9.2. Bounded Buffer

Everyone's favorite example, the bounded buffer smooths out fluctuations in the relative speeds of producers and consumers. It demonstrates implicit receipt of requests.

```
module buffer (producer, consumer : link);
const
   size = whatever;
type
   data = whatever;
var
   buf : array [1..size] of data;
   firstfree, lastfree : [1..size];
```



```
entry put (info : data);
begin
   await firstfree <> lastfree;
   buf[firstfree] := info;
   firstfree := firstfree mod size + 1;
   reply;
end put;

entry get : data;
begin
   await (lastfree mod size + 1) <> firstfree;
   lastfree := lastfree mod size + 1;
   reply (buf[lastfree]);
end get;

begin
   firstfree := 1;
   lastfree := size;
   bind producer to put;
   bind consumer to get;
end buffer.
```

To use the code above, a producer and consumer must actively request the service of the buffer. Such requests are appropriate if both parties know that an intermediary exists. The buffer may be thought of as a "mail-drop." It generalizes easily to serve an arbitrary number of producers and consumers. If the buffer is optional, however, or if it is to be spliced into the connection between an unsuspecting producer/consumer pair, then a different approach is needed. The version below is compatible with the code in section 9.1. The version above was not.

```
module buffer (producer, consumer : link);
const
   size = whatever;
type
   data = whatever;
var
   buf : array [1..size] of data;
   firstfree, lastfree : [1..size];

entry transfer (info : data);
begin
   await firstfree <> lastfree;
   buf[firstfree] := info;
   firstfree := firstfree mod size + 1;
   reply;
end transfer;

begin
   firstfree := 1;
   lastfree := size;
   bind producer to transfer;
   loop
      await (lastfree mod size + 1) <> firstfree;
      connect transfer (buf[lastfree] | ) on consumer;
      lastfree := lastfree mod size + 1;
   end;
end buffer.
```

9.3. Priority Scheduler

The priority scheduler was described in section 2.6.2 of chapter 1. It schedules a resource among a community of clients, highest priority first. Each client calls *schedule_me* to obtain access to the resource. It calls *im_done* to make the resource available to others. For the sake of simplicity, I assume that the priorities of separate clients are distinct. The program is overly simplistic in that

it incorporates no mechanism to recover the resource if a client terminates while holding it.

```
module scheduler (creator, resource : link);

type priority = whatever;
var available : Boolean;

   module priority_queue;
   import
      priority;
   export
      insert, delete, top;

      procedure insert (level : priority);
      begin
         -- add new level to queue
      end insert;

      procedure delete (level : priority);
      begin
         -- remove old level from queue
      end delete;

      function top : priority;
      begin
         -- return highest priority in queue
      end top;

   begin -- priority queue
      -- initialize queue to empty
   end priority_queue;

entry im_done (returned : link); forward;
```

```
entry schedule_me (level : priority) : link;
begin
   insert (level);
   await available and level = top;
   available := false;
   unbind curlink from schedule_me;
   bind curlink to im_done;
   reply (resource);
   delete (level);
end schedule_me;

entry im_done; -- (returned : link);
begin
   unbind curlink from im_done;
   bind curlink to schedule_me;
   available := true;
   resource := returned;
   reply;
end im_done;

entry newclient (client : link);
begin
   reply;
   bind client to schedule_me;
end newclient;

begin
   bind creator to newclient;
   available := true;
end scheduler.
```

9.4. Readers and Writers

The readers/writers problem is well-known and has many variants [33]. The solution presented here avoids starvation of either readers or writers.


```
module readwrite (creator : link);

const maxwriters = whatever;
type ticket = [0..maxwriters];
var
   free, current : ticket;
      -- writers take tickets like the ones at a bakery.
      -- free is the next available number;
      -- current is the one now being served.
   readers, writers, waitingreaders, waitingwriters : integer;
      -- writers is always 0 or 1.

entry doread; -- should have arguments
begin
   -- whatever;
end doread;

entry dowrite; -- should have arguments
begin
   -- whatever;
end dowrite;

entry startread; forward; entry startwrite; forward;
entry endread; forward; entry endwrite; forward;

entry startread;
begin
   if waitingwriters = 0 and writers = 0 then
      readers := readers + 1;
   else
      waitingreaders := waitingreaders + 1;
      await waitingreaders = 0;
   end;
   unbind curlink from startwrite, startread;
   bind curlink to doread, endread;
   reply;
end startread;
```

```
entry startwrite;
var
   turn : ticket;
begin
   if readers = 0 and writers = 0 then
      writers := writers + 1;
   else
      waitingwriters := waitingwriters + 1;
      turn := free; free := free mod maxwriters + 1;
      await current = turn;
   end;
   unbind curlink from startread, startwrite;
   bind curlink to doread, dowrite, endwrite;
   reply;
end startwrite;

entry endread;
begin
   unbind curlink from doread, endread;
   bind curlink to startread, startwrite;
   readers := readers - 1;
   if readers = 0 then
      if waitingwriters <> 0 then
         writers := 1;
         waitingwriters := waitingwriters - 1;
         current := current mod maxwriters + 1;
      end;
   end;
   reply;
end endread;
```



```
entry endwrite;
begin
   unbind curlink from doread, dowrite, endwrite;
   bind curlink to startread, startwrite;
   writers := writers - 1;
   if waitingreaders <> 0 then
      readers := waitingreaders;
      waitingreaders := 0;
   elsif waitingwriters <> 0 then
      writers := 1;
      waitingwriters := waitingwriters - 1;
      current := current mod maxwriters + 1;
   end;
   reply;
end endwrite;

entry newclient (client : link);
begin
   reply;
   bind client to startread, startwrite;
end newclient;

begin  -- initialization
   readers := 0; writers := 0;
   waitingreaders := 0; waitingwriters := 0;
   current := 0; free := 1;
   bind creator to newclient;
end readwrite.
```

Chapter 3

Rationale

1. Introduction

The preceding chapter sketched an overview of LYNX. This chapter explains the rationale behind the design. Some of the features of LYNX are unique; others were chosen from among the possibilities presented by existing languages. Unique features are the result of major design decisions. They are discussed in section 2. Minor decisions are discussed in section 3. The concluding section describes practical experience building servers with LYNX.

2. Major Decisions

Every language is heavily influenced by the perspective of its designer(s). The languages of chapter 1 grew out of efforts to generalize existing sequential languages, first to multiple processes, then to multiple processors. LYNX was approached from an entirely different direction. It *began* with the distributed processes and worked to increase their effectiveness through high-level language support.

Previous languages introduced new models for distributed computation. Aiming for elegance, they attempted to *guess* which small set of concepts would prove to be fundamental. In so doing they often unified concepts that are better kept distinct. By contrast, LYNX *captures* a model that was already in use. It supports the concepts that proved fundamental in the construction of servers for Charlotte. The most important of these concepts are the process, the link, and

the thread of control.

Processes are central; they are what makes distributed programs special. Discussion of LYNX divides naturally into two subtopics: features that support interaction between processes and features that support computation within processes. Links are the key to the former topic; threads of control are the key to the latter.

2.1. Links

Links are a tool for representing distributed **resources**. A resource is a fundamental concept. It is an *abstraction*, defined by the semantics of its external interface and approached conceptually as a single entity. The definition of a resource is entirely in the hands of the programmer who creates it. Examples of resources include open files, query processors, physical devices, data streams, and available blocks of memory. The interface to a resource may include an arbitrary number of remote operations. An open file, for example, may be defined by the semantics of read, write, seek, and close operations.

Recent sequential languages have provided explicit support for data abstraction. Modula modules [115], Ada packages [108], and Clu clusters [78] are obvious examples. Sequential mechanisms for abstraction, however, do not generalize easily to the distributed case. They are complicated by the need to share resources among more than one loosely-coupled process. Several issues are involved.

- Reconfiguration

  Resources move. It must be possible to pass a resource from one process to another and to change the implementation of a resource without the

knowledge of the processes that use it.

- Naming

  A resource needs a single name that is independent of its implementation. Process names cannot be used because a single process may implement an arbitrary number of resources. Operation names cannot be used because a single resource may provide an arbitrary number of operations in its external interface.

- Type Checking

  Operations on resources are at least as complicated as procedure calls. In fact, since resources change location at run time, their operations are as complicated as calls to *formal* procedures. Type checking is crucial. It guarantees that a resource and its users never misinterpret one another.

- Protection

  Even if processes interpret each other correctly, they still cannot *trust* each other. Neither the process that implements a resource nor the process that uses it can afford to be damaged by the other's bad behavior.

In light of these issues, links appear ideally suited to representing distributed resources. As first-class objects they are easily created, destroyed, stored in data structures, passed to subroutines, or moved from one process to another. Their names are independent both of the processes that implement them and the operations they support. A client may hold a link to one of a community of servers. The servers may cooperate to implement a resource. They may pass their end of the client's link around among themselves in order to balance their workload or to connect the client to the member of their group most appropriate for serving its requests at a particular point in time. The client need not even be

aware of such goings on.

Names for links are **uniform** in the sense that there is no need to differentiate, as one must in Ada, between communication paths that are statically declared and those that are accessed through pointers. Moreover, links are *one-one* paths; a server is free to choose the clients with which it is willing to communicate at any particular time. It is free to consider clients as a group by gathering their links together in a set and by binding them to the same entries. It is never forced, however, to accept a request from a arbitrary source that happens to know its address.

Dynamic binding of links to entries is a simple but effective means of providing protection. As demonstrated in the priority scheduler and readers/writers examples of chapter 2 (sections 9.3 and 9.4), bindings can be used to control the access of particular clients to particular operations. With many-one paths no such control is possible. Ada, for example, can only enforce a solution to the readers/writers problem by resorting to a system of keys [113].[8]

The protection afforded by links is not, of course, complete. In particular, though a process can make or break bindings on a link-by-link basis, it has no way of knowing which *process* is attached to the far end of any link. It is not even informed when an end moves. In one sense, a link is like a capability: it allows its holder to request operations on a resource. In another sense, it is a coarser mechanism that requires access lists for fine-grained protection. The rights to specific operations are controlled by servers through bindings; they are

[8] The "solution" in reference [63] (page 11-11) limits each process to one read or write operation per protected session. It does not generalize to applications in which processes gain access, perform a series of operations, and then release the resource.

not a property of links. Links also differ from capabilities in that they can never be copied and can always be moved.

Protection could be increased by distinguishing between the **server end** and the **client end** of a link. The inability of a server to tell when far ends move is after all a direct consequence of link symmetry. If links were asymmetric one could allow the server ends to move without notice, yet require permission (or at least provide notification) when client ends move. Such a scheme has several disadvantages. Foremost among them is its complexity. Two different types of link variable would be required, one to access each type of end. *Connect* would require a link to a server. *Accept*, *bind*, and *unbind* would require a link to a client. *Newlink* would return one link of each type. *Destroy* would take an argument of either type. The semantics of enclosures would depend on which end was enclosed; special rules would apply to the movement of links that connected to servers. Finally, communication between peers (who often make requests of each other) would suddenly require *pairs* of links, one for each direction.

Symmetric links strike a compromise between absolute protection on the one hand and simplicity and flexibility on the other. They provide a process with complete run-time control over its connections to the rest of the world, but limit its knowledge about the world to what it hears in messages. A process can confound its peers by restricting the types of requests it is willing to accept, but the consequences are far from catastrophic. Exceptions are the most serious result, and exceptions can be caught. Even an uncaught exception kills only the thread that ignores it.[9]

[9] Admittedly, a malicious process can serve requests and provide erroneous results. No language can prevent it from doing so.

To a large extent, links are an exercise in **late binding**. Since the links in communication statements are variables, requests are not bound to communication paths until the moment they are sent. Since the far end of a link can be moved, requests are not bound to receiving processes until the moment they are received. Since the set of valid operations depends on outstanding bindings and *accepts*, requests are not bound to receiving threads of control until after they have been examined by the receiving process. Only after a thread has been chosen can a request be bound to the types it must contain. Checks must be performed on a message-by-message basis. (Low-cost techniques are discussed in chapter 4, section 3.3.)

Several of the languages in chapter 1 provide late binding for communication paths. Ada [108], Argus [82, 84], and Mesa [74, 89] provide variables that hold a reference to a process. NIL [27, 105] provides variables that hold a reference to a single operation. SR [4, 5] provides **capabilities** that hold references to a *set* of operations, perhaps in different processes. Each of these languages allows references to be passed in messages. Each checks its types at compile time. To permit such checking, each assigns types to the variables that access communication paths. Variables of different types have incompatible values. By contrast, the dynamic type checking of LYNX has two major advantages:

(1) A process can hold a large number of links without being aware of the types of messages they may eventually carry. A name server, for example, can keep a link to each registered process, even though many such processes will have been created long after the name server was compiled and placed in operation.

(2) A process can use the same link for different types of messages at different times, or even at the same time. A server capable of responding to several radically different types of requests need not create an artificial, and highly complicated, variant record type in order to describe the message it expects to receive.

LYNX type checking also differs from that of previous languages in its use of structural equivalence ([50], p. 92). The alternative, name equivalence, requires the compiler to maintain a global name space for types. Beyond the traditional advantages and disadvantages of each approach [107], two specifically *distributed* concerns motivated the adoption of structural equivalence for LYNX.

(1) A global name space requires a substantial amount of bookkeeping, particularly if it is to be maintained on more than one machine. While the task is certainly not impossible, the relative scarcity of compilers that enforce name equivalence across compilation units suggests that it is not trivial, either.

(2) Compilers that do enforce name equivalence across compilation units usually do so by affixing time stamps to *files* of declarations [89, 108, 117]. A change or addition to one declaration in a file *appears* to modify the others. A global name space for distributed programs can be expected to devote a file to the interface for each distributed resource. Mechanisms can be devised to allow simple *extensions* to an interface [76], but certain enhancements will inevitably invalidate all the users of a resource. In a tightly-coupled program, enhancements to one compilation unit may force the unnecessary recompilation of others. In a loosely-coupled system, enhancements to a process like the file server may force the recompilation

of every program in existence.

The Charlotte implementation of LYNX, described in the following chapter, uses *name* equivalence for types *within* each process. The decision to do so was based primarily on expediency: name equivalence was easier to implement. For the issues that LYNX addresses, the intra-process type checking mechanism is more or less irrelevant.

2.2. Threads of Control

Even on a single machine many processes can most easily be written as a collection of largely independent threads of control. Language designers have recognized this fact for many years (see chapter 1, section 4.1), and have often allowed more than one thread to operate inside a single module and share that module's data. The threads have been designed to operate in simulated parallel, that is, *as if* they were running simultaneously on separate processors with access to a common store.

In Argus [82, 84], StarMod [31, 32], and SR [4, 5] a resource is an isolated module. SR calls such modules **resources**; Argus calls them **guardians** and StarMod calls them **processor modules**. Each module is implemented by one or more **processes**. Semantics specify that the processes execute in parallel, but implementation considerations preclude their assignment to separate physical machines. In effect, the "processes" of Argus, SR, and StarMod are the threads of control of LYNX. Guardians, resources, and processor modules correspond to LYNX processes.

Ada allows data to be shared by arbitrary processes (called **tasks**). It has no notion of modules that are *inherently* disjoint. An Ada implementation must

either simulate shared data across machine boundaries or else specify that only processes that share no data can be placed on separate machines. In either case, language semantics specify that processes execute in parallel.

While simulated parallelism may be aesthetically pleasing, it does not reflect the nature of the underlying hardware. On a single machine, only one thread of control can execute at a time. There is no inherent need for synchronization of simple operations on shared data. By pretending that separate threads can execute in parallel, language designers introduce race conditions that should not even exist; they force the programmer to provide explicit synchronization for even the most basic operations.

In Extended CLU ([79, 80], the predecessor to Argus) and in StarMod, monitors and semaphores are used to protect shared data. These mechanisms are provided in addition to those already needed for inter-module interaction. They lead to two very different forms of synchronization in almost every program.

In Ada and SR, processes with access to common data synchronize their operations with the same message-passing primitives used for inter-module interaction. Small-grain protection of simple variables is therefore rather costly.

Argus sidesteps the whole question of concurrent access with a powerful (and complicated) **transaction** mechanism that serializes even large-grain operations. Programmers have complete control over the exact meaning of atomicity for individual data types [110, 111]. Such an approach may prove ideal for the on-line transaction systems that Argus is intended to support. It is not appropriate for the comparatively low-level operations of operating system servers. Servers might choose to *implement* a transaction mechanism for processes that want one. They must, however, be prepared to interact with arbitrary clients. In

an environment where transactions are not a fundamental concept, servers cannot afford to rely on transactions themselves.

A much more attractive approach to intra-module concurrency can be seen in the semantics of Brinch Hansen's Distributed Processes [22]. Instead of pretending that entry procedures can execute concurrently, the DP proposal provides for each module to contain a single process. The process jumps back and forth between its initialization code and the various entry procedures only when blocked by a Boolean guard. Race conditions are impossible. The comparatively simple **await** statement suffices to order the executions of entry procedures. There is no need for monitors, semaphores, atomic data, or expensive message passing.

An important goal of LYNX is to provide safe and convenient mechanisms that accurately reflect the structure of the underlying system. In keeping with this goal, LYNX adopts the semantics of entry procedures in Distributed Processes, with six extensions:

(1) Messages can be received explicitly, as well as implicitly.

(2) Entry procedures can reply before terminating.

(3) New threads of control can be created locally, as well as remotely.

(4) Blocked threads can be interrupted by exceptions.

(5) A process can accept external requests while waiting for the reply to a request of its own.

(6) Modules and procedures can nest without restriction.

The last extension is, perhaps, the most controversial. As in Ada, it allows the sharing of non-local, non-global data. Techniques for managing the neces-

sary tree of activation records are well understood [16]. They are discussed briefly in section 3.1 of chapter 4 and in section 7 of the appendix. Activation records for any subroutine that may not return before the next context switch must be allocated from a heap. Even the best storage allocator will require more time than is devoted to incrementing the stack pointer in more conventional languages. The allocator will not, however, require more time than is often devoted to saving line numbers and other debugging information when subroutines are called. The automatic management of state for independent conversations is certainly worth at least as much effort as the maintenance of data for post-mortem dumps.

Admittedly, the mutual exclusion of threads in LYNX prevents race conditions only between context switches. In effect, LYNX code consists of a series of critical sections, separated by blocking statements. Since context switches can occur inside subroutines, it is not even immediately obvious where those blocking statements are. The compiler can be expected to help to some extent by producing listings in which each (potentially) blocking statement is marked. Experience to date has not uncovered a serious need for inter-thread synchronization across blocking statements. For those cases that do arise, a simple Boolean variable in an await statement performs the work of a semaphore.

3. Minor Decisions

3.1. Synchronization

As described in section 2.4 of chapter 1, there are three principal approaches to message synchronization.

(1) No-Wait Send

A sender continues execution immediately, even as its message is beginning the journey to wherever it is going.

(2) Synchronization Send

The sender waits until the message has been received before continuing execution.

(3) Remote-Invocation Send

The sender waits until it receives a reply from the receiver.

The principal advantage of the no-wait send is a high degree of concurrency. The principal disadvantages are the complexity of buffering messages and the difficulty of reflecting errors back to a sender who may have proceeded an arbitrary distance past the point of call. For LYNX, the concurrency advantage is not as compelling as it might first appear, since a process can continue with other threads of control when a given one is blocked, and since node machines may be multiprogrammed anyway. The disadvantage of buffering is not particularly compelling either. It makes the run-time support package larger and more complicated, and it necessitates flow control, but solutions do exist. The deciding factor is the problem of error reporting. Unlike traditional i/o (which often *is* implemented in a no-wait fashion), interprocess message passing involves type-checked communication with potentially erroneous or even malicious user programs. The likelihood of errors is high, as is the need to detect and cope with them in a synchronous fashion.

LYNX provides remote-invocation send rather than synchronization send because it is a more powerful mechanism and because it requires fewer underlying messages in common situations. Synchronization send does overcome the

disadvantages of the no-wait send, but it requires a top-level acknowledgment. The acknowledgment cannot be sent by the operating system because it contains confirmation of the correctness of types. Given the ubiquity of client/server relationships, it is reasonable to expect most messages to be requests that need explicit replies. As long as an acknowledgment is being sent anyway, it might as well carry useful data. Synchronization send is easily simulated with remote invocation by sending an immediate reply. Simulating remote invocation with synchronization send requires extra messages.

There is some motivation for providing synchronization send *in addition* to remote invocation. For messages that need no reply, top-level acknowledgments can be sent by run-time support routines, rather than by the user's program, allowing the sender to be unblocked after two fewer context switches on the receiving end. The savings are too small, however, to justify cluttering the language with a second kind of send.

3.2. Explicit and Implicit Message Receipt

LYNX provides two very different means of receiving messages: the accept statement and the mechanism of bindings. The former allows messages to be received *explicitly*; the latter allows them to be received *implicitly*. Each has applications for which it is appropriate and others for which it is awkward and confusing. A practical language needs both.

Implicit receipt most accurately reflects the externally-driven nature of most servers:


```
module server; -- implicit receipt
var client : link;

entry A;
begin
  - - -
end A;

entry B;
begin
  - - -
end B;

begin
  bind client to A, B;
end server.
```

Explicit receipt is much less straightforward:

```
module server; -- explicit receipt
type
  message = record
    case class : whatever of
      - - -
    end;
  end;
var
  client : link;
  m : message;

entry request (m : message); remote;
```

```
begin  -- server
  loop
    accept request (m) on client;
      case m.class of
        {A}
          ...
        {B}
          ...
      end;
    reply;
  end;
end server.
```

In these examples the explicit approach has several disadvantages:

(1) It requires a complicated variant record structure to reflect differences in arguments between the different entries.

(2) It is *misleading*. The server module is a *passive* body of code; it does nothing until called from outside.

(3) It carries implications about concurrency that may not be appropriate.

The third disadvantage is probably the most important. In the second example above, the server cannot begin work on a new request until the previous one has finished. If a request requires communication with some third party, the server will be blocked (needlessly) until that communication completes. It is of course possible within the server to assign a different (active) thread of control to each type of request, but the server will still be limited to one partially-completed request of each type. There is in general no way to accommodate an unspecified number of requests with a fixed number of threads.

Explicit receipt is most useful for the exchange of messages between active, cooperating peers. Its use was demonstrated by the producer and consumer of chapter 2, section 9.1. In this case the implicit approach is considerably less

attractive:

```
module producer (consumer : link);

type data = whatever;
entry newstuff (info : data); remote;

function produce : data;
begin
  -- whatever
end produce;

begin  -- producer
  loop
    connect newstuff (produce | ) on consumer;
  end;
end producer.
```

```
module consumer (producer : link);  -- implicit receipt

type data = whatever;
var
  buffer : data;
  consumed, received : Boolean;

entry newstuff (info : data);  -- called by producer
begin
  await consumed;
  buffer := info;
  received := true;
end newstuff;

procedure consume (info : data);
begin
  -- whatever
end consume;
```

```
begin  -- consumer
   consumed := true; received := false;
   loop
      await received;
      consume (buffer);
      consumed := true;
   end;
end consumer.
```

The "newstuff" entry really belongs in the consumer's main loop. Moving it out-of-line results in code that is unnecessarily complicated, asymmetric, and hard to understand. The dual solution [15] is equally bad:

```
module producer (consumer : link); -- implicit receipt

type data = whatever;
var
   buffer : data;
   produced, sent : Boolean;

entry oldstuff : data; -- called by consumer
begin
   await produced;
   reply (buffer);
   sent := true;
end oldstuff;

function produce : data;
begin
   -- whatever
end produce;
```



```
begin  -- producer
   produced := false; sent := true;
   loop
      await sent;
      buffer := produce;
      produced := true;
   end;
end producer.
```

```
module consumer (producer : link);

type data = whatever;
entry oldstuff : data; remote;

procedure consume (info : data);
begin
   -- whatever
end consume;

var buffer : data;

begin  -- consumer
   loop
      connect oldstuff ( | buffer) on producer;
      consume (buffer);
   end;
end consumer.
```

Some existing languages, notably StarMod [31, 32], already provide both explicit and implicit receipt. LYNX goes one step farther by allowing a process to decide at run time which form(s) to use when, and on which links.

3.3. Syntax

The accept statement in LYNX is designed to be as simple as possible. It does not, for example, define a nested scope the way it does in Ada. There is no need to copy message parameters into variables that would remain visible after the scope was closed. In addition, instead of declaring parameter types at each accept statement, LYNX allows *accept*s at more than one place in the code to share declarations. Since entry headers already declare parameter types for implicit receipt, it seems reasonable to use them for explicit receipt as well. An operation that is served both explicitly and implicitly need only be declared once. An operation that is only served explicitly can be declared with an entry whose body is *remote*.

Entry headers segregate their request parameters and reply types. An obvious alternative would use a single parameter list, with names for *all* parameters. Request, reply, and request/reply parameters could appear in any order. Unfortunately, the resulting syntax would not reflect the structure of accept statements. It would also tend to hide the underlying request and reply messages, messages that I prefer to keep visible.

Unlike most proposed languages with explicit receipt, LYNX does not provide a mechanism for accepting a message on any one of a *set* of links. Applications examined to date appear to need such non-determinism only in cases where implicit receipt is the more appropriate approach. A non-deterministic version of explicit receipt would be a fairly straightforward addition to the language, should it prove necessary.

3.4. Exceptions

I am aware of no precedent for the semantics of user-defined exceptions in LYNX. Since built-in exceptions were needed anyway, the addition of the user-defined variety allowed one thread to interrupt another with little extra syntax. Interruptions are useful in protocols where one thread may discover that the communication for which another thread is waiting is no longer appropriate (or possible). One example is a stream-based file server. The code below sketches the form that such a server might take.

The file server begins life with a single link to the **switchboard**, a name server that introduces clients to various other servers. When the switchboard sends the file server a link to a new client (line 49), the file server binds that link to an entry procedure for each of the services it provides. One of those entries, for opening files, is shown in the code below (lines 3-48).

Open files are represented by links. Within the server, each file link is managed by a separate thread of control. New threads are created in response to *open* requests. After verifying that its physical file exists (line 24), each thread creates a new link (line 25) and returns one end to its client. It then binds the other end to appropriate sub-entries. Among these sub-entries, context is maintained automatically from one request to the next. As suggested by Black [15], bulk data transfers are initiated by the producer (with *connect*) and *accept*ed by the consumer. When a file is opened for writing the server plays the role of consumer. When a file is opened for reading the server plays the role of producer. Seek requests are handled by raising an exception (line 21, caught at line 37) in the file-server thread that is attempting to send data out over the link. Clients close their files by destroying the corresponding links.

```
1   module fileserver (switchboard : link);
2   type string = whatever; bytes = whatever;

3   entry open (filename : string; readflag, writeflag, seekflag : Boolean)
4       : link;
5   var filelnk : link; readptr, writeptr : integer;
6   exception seeking;

7       procedure put (data : bytes; filename : string; writeptr : integer);
8           external;
9       function get (filename : string; readptr : integer) : bytes; external;
10      function available (filename : string) : Boolean; external;

11      entry writeseek (fileptr : integer);
12      begin
13          writeptr := fileptr; reply;
14      end writeseek;

15      entry stream (data : bytes);
16      begin
17          put (data, filename, writeptr); writeptr := writeptr + 1; reply;
18      end stream;

19      entry readseek (newptr: integer);
20      begin
21          readptr := newptr; raise seeking; reply;
22      end readseek;

23  begin  -- open
24      if available (filename) then
25          reply (newlink (filelnk));  -- release client
26          readptr := 0; writeptr := 0;

27          if writeflag then
28              if seekflag then bind filelnk to writeseek; end;
29              bind filelnk to stream;
30          end;
```



```
31          if readflag then
32              if seekflag then bind filelnk to readseek; end;
33              loop
34                  begin
35                      connect stream (get (filename, readptr) | ) on filelnk;
36                      readptr := readptr + 1;
37                  when seeking do
38                      -- nothing; continue loop
39                  when filelnk REMOTE_DESTROYED do
40                      exit;  -- leave loop
41                  end;
42              end;  -- loop
43          end;  -- if readflag
44      else  -- not available
45          reply (nolink);  -- release client
46      end;
47      -- control will not leave 'open' until nested entries have died
48  end open;

49  entry newclient (client : link);
50  begin
51      bind client to open; reply;
52  end newclient;

53  begin  -- main
54      bind switchboard to newclient;
55  end fileserver.
```

4. Experience

Original implementations of the Charlotte servers were written in Modula ([40], sequential features only) with direct calls to the Charlotte primitives. Work is underway to re-build the servers in LYNX. As of this writing, the file server is finished and the memory manager (the *starter*) is well under way. The first version of the terminal driver (a new server) is being written in LYNX as well.

Several preliminary conclusions can be drawn.

- LYNX programs are considerably easier to write than their sequential counterparts. The Modula fileserver was written and re-written several times over a period of about two years. It has been a constant source of trouble. The LYNX fileserver was written in two weeks. It would have required even less time had the LYNX run-time package already been debugged.
- The source for LYNX programs is considerably shorter than equivalent sequential code. The new fileserver is just over 300 lines long. The original is just under 1000 lines.[10]
- LYNX programs are considerably easier to read than their sequential counterparts. While this is a highly subjective measure, it appears to reflect the consensus of programmers who have examined both versions.

[10] Object code from LYNX is somewhat longer than its sequential counterpart. See section 6.1 of chapter 4.

Chapter 4

Implementation

1. Introduction

This chapter describes two implementations of LYNX. The first, for the Charlotte distributed operating system, was actually built. The second, for an operating system called SODA, was designed on paper only.

The Charlotte implementation took just under two years of part-time work by a single programmer. It required the development of several interesting techniques, particularly for type checking and for moving large numbers of links. It also encountered unexpected problems with the Charlotte kernel interface. Surprisingly, though the design of LYNX was based largely on the primitives provided by Charlotte, the SODA implementation is in some ways considerably simpler.

2. Overview of Charlotte

Charlotte [7, 41] runs on the Crystal multicomputer [34], a collection of 20 VAX 11/750 **node** machines connected by a 10-Mbit/second token ring from Proteon Corporation. Each node has two or three megabytes of memory. Some nodes have disks. Several larger VAXen are used for program development. They run UNIX and are included in the ring.

The Charlotte **kernel** is replicated on each node. It provides direct support for both **processes** and **links**. Its most important system calls are the following.

MakeLink (var end1, end2 : link)

Create a link and return references to its ends.

Destroy (myend : link)

Destroy the link with a given end.

Send (L : link; buffer : address; length : integer; enclosure : link)

Start a **send activity** on a given link end, optionally enclosing one end of some other link.

Receive (L : link; buffer : address; length : integer)

Start a **receive activity** on a given link end.

Cancel (L : link; d : direction)

Attempt to cancel a previously-started send or receive activity.

Wait (var e : description)

Wait for an activity to complete, and return its description (link end, direction, length, enclosure).

All calls return a status code. All but *Wait* are guaranteed to complete in a bounded amount of time. *Wait* blocks the caller until an activity completes.

The kernel matches send and receive activities. It allows only one outstanding activity in each direction on a given end of a link. Completion must be reported by *Wait* before another similar activity can be started. Buffers are managed by user processes, in user address space. Results are unpredictable if a process uses a buffer between the starting of an activity and the notification of its completion.

3. The Charlotte Implementation

The Charlotte LYNX compiler consists of about 13000 lines of Pascal source and about 400 lines of C. (The C code includes initialized data structures, UNIX I/O, and logical operations on bit fields.) The run-time system for LYNX consists of about 5000 lines of C and 200 lines of assembler. The compiler uses a table-driven scanner and LL(1) parser, with FMQ [45] syntactic error correction. It generates error-free C source code peppered with escapes to assembly language. The standard C compiler is a second pass. The result is a friendly but slow-running utility that produces code of acceptable quality. The appendix contains a thorough description of the implemented language.

3.1. Threads of Control

Lexical nesting of entries implies the sharing of non-local, non-global data among multiple threads of control and precludes stack-based storage allocation for routines that may cause context switches. For the sake of efficiency, the compiler notices any routines that cannot cause context switches and allocates their activation records on a stack. For the rest of the routines, the storage allocator uses a naive first-fit algorithm on a heap. A production-quality implementation would probably require experimentation with other allocators.

The LYNX implementation uses a static chain instead of a display. The only context that must be saved when control switches from one thread to another is the contents of ten of the VAX's general registers and the value of a global pointer to the context block of the current thread. Each context block contains several Boolean flags, links for the queue on which its thread resides, and room for a subroutine-call stack mark. A thread yields control by putting itself on an appropriate queue, aiming the stack pointer at its context block, and calling a

procedure known as the **dispatcher**. The subroutine call instruction saves the appropriate registers.

The dispatcher maintains queues of threads that are blocked for various reasons. When called, it decides which thread to run next, aims the VAX frame pointer at the thread's context block, and executes a return-from-subroutine instruction.

The threads blocked at await statements reside on a single circular queue. Each time it is called, the dispatcher returns control to the next thread on that queue. The resumed thread re-evaluates its Boolean condition and returns immediately to the dispatcher if that condition is not true. If the dispatcher discovers that it has come all the way around the queue without finding a true condition, then it executes a Wait system call to obtain notification of a completed activity from the Charlotte kernel. The Boolean expressions in every await statement are therefore re-evaluated every time a message is sent or received. The cost involved is discussed in section 6.2.

3.2. Communication

The run-time system uses Charlotte links to implement the links of LYNX. Link variables are indices into a table invisible to the user. For each link end, the table contains

- head pointers for the queues of threads waiting to send and receive messages on the link,
- the size and address of the buffer of the outstanding receive activity (if any),
- a sequence counter for connect statements (used to detect unwanted reply messages), and

- a small amount of status information.

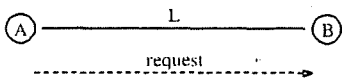
For a connect statement, the run-time system starts a send activity with the kernel. When that activity completes, it starts a receive activity. Run-time routines also start send activities for reply statements and receive activities for accept statements and bindings.

There are two basic kinds of messages: **requests** and **replies**. Requests are caused by connect statements. Replies are caused by reply statements. Each message includes several bytes of self-description. For requests, this information consists of the name of the remote operation, the identity of the thread (the *client*) that executed the connect statement, a sequence number, and a code word for type-checking. For replies, only the sequence number and client thread are specified. They are copied from the request message.

The operation name is used to direct an incoming request to an appropriate thread of control (a *server*). The identity of the client is used to direct an incoming reply. Sequence numbers allow the dispatcher to detect replies for clients that have felt exceptions and are no longer waiting. The type-checking information is discussed in the following section.

3.3. Type Checking

As explained in chapter 3, message type-checking in LYNX is based on structural equivalence. Equivalent types have the same **canonical description**. In the absence of pointers, these descriptions can mimic ordinary type declarations, with the subtypes expanded in-line. Canonical descriptions for messages can then be built from the descriptions of their parameters.



A thread in **B** receives the request and begins serving the operation. **A** now expects a reply on **L** and posts a *Receive* with the kernel. Now suppose some other thread in **B** requests an operation on **L**. **A** will receive **B**'s request before the reply it wanted. Since **A** may not be willing to serve requests on **L** at this point, **B** is not able to assume that its request is being served simply because **A** received it.

A similar problem arises if **A** binds **L** to a number of entries and then breaks the bindings before all its threads are blocked. In the interests of concurrency, the run-time support routines will have posted a *Receive* with the kernel as soon as the first binding was made. When the last one is broken, they will attempt to cancel the *Receive*. If **B** has requested an operation in the meantime, the *Cancel* will fail. The next time **A** calls *Wait*, it will receive notification of the message from **B**, a message it does not want. Delaying the start of receive activities until all threads are blocked does not help. **A** must still start activities for *all* the messages it would be willing to receive. It will continue execution after *one* of them completes. Before waiting for a second, it may change the set of messages it is willing to receive.

The first problem arises because Charlotte provides no way to screen messages within a single link. The second problem arises because Charlotte provides no way for the user program to say "please receive exactly one request, on any one of the following set of link ends." The second problem would be more obvious if LYNX provided a non-deterministic version of explicit receipt, in the style

of CSP or Ada (see chapter 1, section 2.6.1, and chapter 3, section 3.3).

In both cases it is tempting to let **A** buffer unwanted messages until it is again willing to receive from **B**, but such a solution is impossible for two reasons. First, a user-defined exception may arise in **B**, causing it to attempt to cancel the *Send* on **L**. Since **A** does not yet want the message, the *Cancel* should succeed, but cannot. Second, the scenario in which **A** receives a request but wants a reply can be repeated an arbitrary number of times, and **A** cannot be expected to provide an arbitrary amount of buffer space.

A must return unwanted messages to **B**. In addition to the request and reply messages needed in simple situations, we now introduce the **retry** message. *Retry* is a negative acknowledgment. In the second scenario above, when **A** has broken all its bindings, a re-sent message from **B** will be delayed by the kernel, since **A** will have no *Receive* outstanding. To prevent arbitrary numbers of retransmissions in the first scenario (since **A** will keep a *Receive* posted for the reply it wants), we also introduce the **forbid** and **allow** messages. *Forbid* denies a process the right to send requests. (It is still free to send replies.) *Allow* restores that right. *Retry* is equivalent to *forbid* followed by *allow*. Both *forbid* and *retry* return any link end that was enclosed in the unwanted message. A process that has received a forbid message keeps a *Receive* posted on the link in hopes of receiving an allow message.[11] A process that has sent a forbid message remembers that it has done so and sends an allow message as soon as it is either willing to receive requests or else has no *Receive* outstanding (so the kernel will delay all messages).

[11] This of course makes it vulnerable to receiving unwanted messages itself.

The size of the description of a message depends on the number of parameters and on the complexity of their types. Rather than enclose the description itself in every message, the LYNX implementation uses a hash function to reduce canonical descriptions to a single word. An obvious hash function treats a string of symbols as an integer base N, where N is the size of the symbol set.

Suppose $<a> = a_{n-1}\, a_{n-2}\, a_{n-3} \cdots a_0$ is a string of symbols. Let

$$hash(<a>) =_{def} \left(\sum_{i=0}^{n-1} N^i \, ord(a_i) \right) \textbf{ mod } p .$$

If $<a>$ is the canonical description of a type A, we say

$$hashval(A) = hash(<a>) \quad \text{and} \quad hashlen(A) = n .$$

In the Charlotte LYNX implementation, N is 37 and p is $2^{32} - 5 = 4294967291$. Symbols are represented by the values 1-37. No symbol has value 0, since prepending a zero-value symbol to a string would not change its hash code. The lack of a zero-value symbol allows N to be used for the value of the final symbol without introducing ambiguity.

In addition to simplicity, this hash function has the advantage of incremental computation. It obviates the need to store explicit canonical descriptions. The compiler remembers the hash code and length of each type's canonical description, but not the description itself. When a composite type is declared, its hash code and description length can be calculated from those of its constituent types.

Suppose, for example, we are given the following declarations:


```
A =  [1..10];
B =  record
         i, j : integer;
     end;
C =  array A of B;
```

We would like the hash code for C to be the same as the hash code for

```
C' = array [1..10] of record
         i, j : integer;
     end;
```

This is precisely the result we obtain by letting

$$hashval(C) = \left[a \times N^{hashlen(A)} + hashval(A) \right] \times N^{hashlen(B)} + hashval(B) .$$

$$hashlen(C) = 1 + hashlen(A) + hashlen(B),$$

where a is the value of the symbol "array" as an N-ary digit. All arithmetic is carried out in the ring of integers mod p.

As currently defined, LYNX provides no pointers. If it did, the hashing technique would need to be changed. The problem stems from the need for forward references in defining circular structures. When a given type is first encountered we might not know the nature of its parts. We could still derive a canonical description and hash code for each type, but we could not do it incrementally the way we could above.

At the end of each declaration section all existing types will be fully defined. Well-known techniques can be used to determine which are structurally distinct and which equivalent [72]. Symbol-table entries for equivalent types can be coalesced. We can then use the string-of-symbols notation, augmented with backpointers, to construct canonical descriptions for the types that remain. We expand each declaration recursively until we encounter a cycle. We then insert a

backpointer to the point where the cycle began. For example, the type

```
sequence = record
               item : integer;
               next : ^sequence;
           end;
```

might be represented by the string "**record integer pointer -3 end.**"

In a language with external compilation, the hashing of type descriptions provides a simple mechanism for type checking across compilation units with ordinary linkage editors [99]. For messages, it reduces the overhead of run-time type checking to a simple one-word comparison. The only disadvantage of the scheme is its lack of absolute security. With a reasonable word size, however, the range of the hash function will include enough values to make errors extremely unlikely.

In the Charlotte LYNX implementation, operation names are also hashed for inclusion in the self-descriptive part of messages. The hash function is the same as for types. The letters, digits, and underscore are assigned the values 1-*N*. The compiler checks to make sure that no process uses the same hash code for more than one accept statement or non-remote entry.

3.4. Exceptions

The routines that manage exceptions treat each group of when clauses as a single handler. Each scope keeps the address of its innermost handler group in its stack frame. When an exception occurs, the stack of the affected thread is searched until a handler is found. The compiler inserts a dummy handler at the end of every entry, so the search never proceeds back through a fork in the run-time environment tree. The code for a group of *when* clauses simply re-raises

exceptions it is not supposed to handle.

Built-in exceptions are raised in a single thread. User-defined exceptions may be raised in many threads at once. Every user-defined exception has an associated list of threads with appropriate handlers. Threads insert and remove themselves from the list as they enter and leave scopes in which the handlers are defined.

Each scope also keeps a list of nested threads. These threads are aborted if an exception escapes the scope when it has no handler.

The run-time package includes exception handlers for each of the communication routines. These handlers are responsible for cleaning up unfinished communication when an exception occurs. Among other things, they often deallocate unused buffers and move their thread from one queue to another.

4. Problems

Despite the fact that much of the design of LYNX was motivated by the primitives of Charlotte, the actual process of implementation proved to be quite difficult. Most of the difficulty stemmed from two sources: the arrival of unwanted messages and the enclosure of ends of more than one link.

4.1. Unwanted Messages

For the vast majority of operations, only two Charlotte messages will be used: one for the request and one for the reply. Unfortunately, complications arise in a number of special cases.

Suppose that a thread in process A requests a remote operation on link L.

Further complications arise when the buffer supplied to *Receive* is too small to hold the message that arrives. This can happen whenever a process is interested in more than one kind of message on a given link. If the messages require different size buffers, the run-time support routines may post a *Receive* with the kernel for the smaller size message before learning that the larger one exists. The larger message may arrive before the *Receive* can be cancelled and restarted with a larger buffer.

Fortunately, Charlotte informs both the sender and the receiver when overflow occurs. The sender's communication routine can retransmit the message on the assumption that the new buffer will be larger. It need only retransmit once; the receiver will not be notified of the overflow until it blocks, and by then it will know the size of the largest valid message. When it posts a second *Receive* its buffer will reflect that size.

If a reply message overflows twice in succession then the server thread can be sure that the client thread has felt an exception and died. The server can continue. If a request message overflows twice in a row, however, the client cannot make a similar assumption. It must wait for a message from the server. If the server is only willing to receive replies, it returns a forbid message. If it is willing to receive requests (but only small ones, presumably), it instructs the client to raise an exception of class INVALID_OP in the thread that sent the request.

A receiver saves the enclosure from a message that overflows. It remembers that enclosure when it receives the retransmission (retransmissions can be recognized by their self-descriptive information). Since a client thread may feel an exception and die before sending a retransmission, the receiver must destroy the saved enclosure if it receives an original (non-retransmitted) message

first. In the absence of exceptions, each sender guarantees that the original and retransmitted versions of a message make consecutive use of their link, with nothing in between.

4.2. Moving Multiple Links

Most data can be transmitted from one machine to another as a simple stream of bytes. Links cannot. Two problems arise. First, since a considerable amount of state information is associated with a link, and since rules forbid moving an active end, the run-time system must be aware of all the references to links contained in a given message. Second, since Charlotte permits only one link end to be enclosed in each message supplied to the kernel, the run-time system must packetize its higher-level messages.

To minimize the size of object files, the Charlotte LYNX compiler leaves all the work of message passing to run-time support routines. At compile time, it constructs descriptors for the request and reply messages of each entry. These descriptors list the offsets within the messages at which references to links may be found. For the following entry,

entry foo (a, b : integer, l : link; c : char) : link, link; ...

the request descriptor would be **<8>** (integers are four bytes long). The reply descriptor would be **<0, 4>**.

Complications arise for sets of links, for arrays containing links, and for variant records. Sets are handled by preceding their offsets with a flag.

set offset

Arrays are handled by embedding a loop in the descriptor.

loop, offset1, offset2, ..., offsetN *endloop*, count, add, backup

Loop and *endloop* are flags. **Count** is the number of elements in the array. **Add** is the size of a single element. **Backup** is the distance back to the *loop* flag. The offsets between the *loop* and *endloop* flags give the locations of links in the first element of the array. Later elements are handled by adding multiples of **add** to those offsets.

Arrays can nest. The routine that interprets message descriptors counts the number of times it has gone around the loop for each level of nesting. It also keeps track of the cumulative **add** correction for offsets. At the end of each array it reduces that correction by **count** times **add**.

Variant records are handled by embedding an if statement in the descriptor.

if, offset, *singleton*, value, [offset, ...,]
range, lvalue, hvalue, [offset, ...,]
singleton, value, *range*, lvalue, hvalue, [offset, ...,]
..., *endif*

The offset following the *if* flag gives the location of the tag of the variant. The lists of singletons and ranges give the valid tag values for the arms of the variant. Each arm may contain nested arrays or other variants.

The value of a link variable is an index into a process-specific table. When enclosed in a message, the value must be changed. Before transmitting a message buffer, the run-time routines change invalid links to *nolink*. Valid links are left alone.[12] On the receiving end, the old values are used to 1) distinguish valid links from *nolink*, and 2) detect duplicates. Once the receiver has examined the buffer, both processes can tell how many distinct, valid links were meant to be enclosed.

[12] It is possible of course that an uninitialized link variable or an old link variable whose value has been reused will be interpreted as a valid reference. This is the standard problem with dangling references.

A request or reply that contains more than one enclosure must be broken into several Charlotte messages. The first packet contains the message buffer and the first enclosure. Additional enclosures are passed in empty **enc** messages (see figure 4.1). For requests, the receiver must return an explicit **goahead** message after the first packet so the sender can tell the request is wanted. No *goahead* is needed for requests with zero or one enclosures, and none is needed for replies,

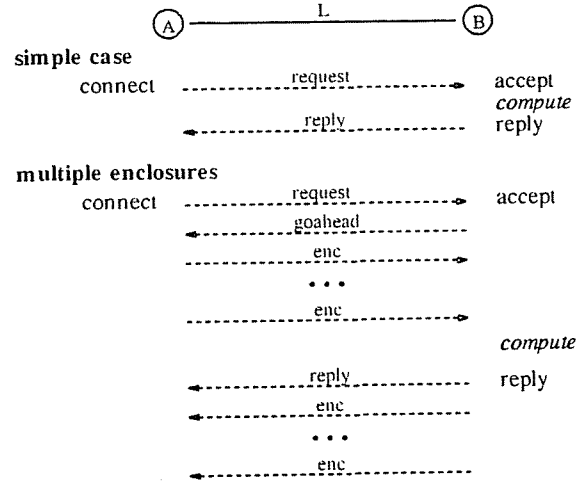


figure 4.1: link enclosure protocol

since a reply is always wanted.

4.3. Semantic Complications

In two instances, the problems described above forced me to do what a "real-life" implementor could not: change the semantics of the language.

(1) I had hoped to allow exceptions to abort unstarted communication as if it had never been requested. This proved impossible, because enclosures may be lost. Consider the following chain of events.

a) Process **A** sends a request to process **B**, enclosing a link end.

b) **B** receives the request unintentionally; it only wanted a reply.

c) **A** feels a local, user-defined exception, aborting the request.

d) **B** crashes before it can send the enclosure back to **A** in a forbid message.

The enclosure is lost, though the semantics of LYNX say the communication never started. Thus the rule: all link ends contained in a message are lost as soon as the communication is requested, whether it finishes or not.

(2) Exceptions of class EXC_REPLY are raised at connect statements when server threads die during rendezvous. I had hoped to define a similar exception class to be raised at reply statements when clients die during rendezvous. In Charlotte, however, that would have required an explicit, top-level acknowledgement from client to server when a reply message was successfully received. The resulting 50% increase in underlying message traffic for typical cases would have been an unacceptable burden. Thus the rule: if a reply statement completes successfully, the server thread can assume the reply was delivered only if the client thread was still alive.

5. A Paper Implementation for SODA

It is worth considering whether the complexity of the implementation just described is the fault of Charlotte or of LYNX. For purposes of comparison, this section describes an implementation of LYNX for SODA, a "Simplified Operating system for Distributed Applications." SODA was designed by Jonathan Kepecs as a part of his Ph. D. research [69].

5.1. A Review of the SODA Primitives

SODA was described in section 4.3.2 of chapter 1. Features needed by the LYNX implementation are summarized here.

Every SODA process has a unique **id**. It also **advertises** a collection of **names** to which it is willing to respond. There is a kernel call to generate new names, unique over space and time. The **discover** kernel call uses unreliable broadcast in an attempt to find a process that has advertised a given name.

Processes do not necessarily *send* messages, rather they **request** the transfer of data. A process that is interested in communication specifies a name, a process id, a small amount of out-of-band information, the number of bytes it would like to send and the number it is willing to receive. Since either of the last two numbers can be zero, a process can request to send data, receive data, neither, or both. The four varieties of request are termed **put**, **get**, **signal**, and **exchange**, respectively.

Processes are informed of interesting events by means of software interrupts. Each process establishes a single **handler** which it can close temporarily when it needs to mask out interrupts. A process feels a software interrupt when its id and one of its advertised names are specified in a request from some other

process. The handler is provided with the id of the requester and the arguments of the request, including the out-of-band information. The interrupted process is free to save the information for future reference.

At any time, a process can **accept** a request that was made of it at some time in the past. When it does so, the request is completed (data is transferred in both directions simultaneously), and the requester feels a software interrupt informing it of the completion and providing it with a small amount of out-of-band information from the accepter. Like the requester, the accepter specifies buffer sizes. The amount of data transferred in each direction is the smaller of the specified amounts.

Completion interrupts are queued when a handler is busy or closed. Requests are delayed; the requesting kernel retries periodically in an attempt to get through (the requesting user can proceed). If a process dies before accepting a request, the requester feels an interrupt that informs it of the crash.

5.2. A Different Approach to Links

A link in SODA can be represented by a pair of unique names, one for each end. A process that owns an end of a link advertises the associated name. The following algorithm can be used to keep track of the location of links and to move their ends from one process to another.

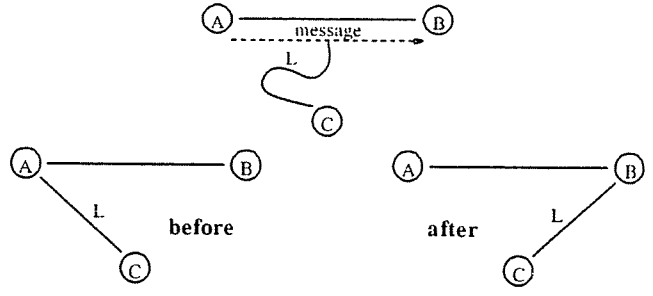
Every process knows the names of the link ends it owns. Every process keeps a **hint** as to the current location of the far end of each of its links. The hints can be wrong, but are expected to work most of the time.

A process that wants to send a LYNX message, either a request or a reply, initiates a SODA *put* to the process it thinks is on the other end of the link. A

process moves link ends by enclosing their names in a message. When the message is SODA-accepted by the receiver, the ends are understood to have moved. Processes on the fixed ends of moved links will have incorrect hints.

A process that wants to receive a LYNX message, either a request or a reply, initiates a SODA *signal* to the process it thinks is on the other end of the link. The purpose of the signal is allow the aspiring receiver to tell if its link is destroyed or if its chosen sender dies. In the latter case, the receiver will feel an interrupt informing it of the crash. In the former case, we require a process that destroys a link to accept any previously-posted status *signal* on its end, mentioning the destruction in the out-of-band information. We also require it to accept any outstanding *put* request, but with a zero-length buffer, and again mentioning the destruction in the out-of-band information. After clearing the *signal*s and *put*s, the process can **unadvertise** the name of the end and forget that it ever existed.

Suppose now that process **A** has a link **L** to process **B** and that it sends its end to process **C**.



If **C** wants to send or receive on **L**, but **B** terminates after receiving **L** from **A**, then **C** must be informed of the termination so it knows that **L** has been destroyed. **C** will have had a SODA request posted with **A**. **A** must accept this request so that **C** knows to watch **B** instead. We therefore adopt the rule that a process that moves a link end must accept any previously-posted SODA request from the other end, just as it must when it destroys the link. It specifies a zero-length buffer and uses the out-of-band information to tell the other process where it moved its end. In the above example, **C** will re-start its request with **B** instead of **A**.

The amount of work involved in moving a link end is very small, since accepting a request does not even block the accepter. More than one link can be enclosed in the same message with no more difficulty than a single end. If the fixed end of a moving link is not in active use, there is no expense involved at all. In the above example, if **C** receives a SODA request from **B**, it will know that **L** has moved.

The only real problems occur when an end of a dormant link is moved. If our example, if **L** is first used by **C** after it is moved, **C** will make a SODA request of **A**, not **B**, since its hint is out-of-date. There must be a way to fix the hint. If each process keeps a cache of links it has known about recently, and keeps the names of those links advertised, then **A** may remember it sent **L** to **B**, and can tell **C** where it went. If **A** has forgotten, **C** can use the *discover* command in an attempt to find a process that knows about the far end of **L**.

A process that is unable to find the far end of a link must assume it has been destroyed. If **L** exists, the heuristics of caching and broadcast should suffice to find it in the vast majority of cases. If the failure rate is comparable to

that of other "acceptable" errors, such as garbled messages with "valid" checksums, then the heuristics may indeed be all we *ever* need.

Without an actual implementation to measure, and without reasonable assumptions about the reliability of SODA broadcasts, it is impossible to predict the success rate of the heuristics. The SODA *discover* primitive might be especially strained by node crashes, since they would tend to precipitate a large number of broadcast searches for lost links. If the heuristics failed too often, a fall-back mechanism would be needed.

Several absolute algorithms can be devised for finding missing links. Perhaps the simplest looks like this:

- Every process advertises a **freeze** name. When **C** discovers its hint for **L** is bad, it posts a SODA request on the freeze name of every process currently in existence (SODA makes it easy to guess their ids). It includes the name of **L** in the request.
- Each process accepts a freeze request immediately, ceases execution of everying but its own searches (if any), increments a counter, and posts an **unfreeze** request with **C**. If it has a hint for **L**, it includes that hint in the freeze accept or the unfreeze request.
- When **C** obtains a new hint or has unsuccessfully queried everyone it accepts the unfreeze requests. When a frozen process feels an interrupt indicating that its unfreeze request has been accepted or that **C** has crashed, it decrements its counter. If the counter hits zero, it continues execution. The existence of the counter permits multiple concurrent searches.

This algorithm has the considerable disadvantage of bringing every LYNX process in existence to a temporary halt. On the other hand, it is simple, and should

only be needed when a node crashes or a destroyed link goes unused for so long that everyone has forgotten about it.

5.3. Comparison to Charlotte

The SODA implementation avoids the major problems of Charlotte. It moves multiple links in a single message. It receives no unwanted messages. It always knows what size buffer to allocate. It needs no retry, forbid, or allow messages. It does not require the semantic compromises of section 4.3.

In all fairness, however, there are potential problems that bear mentioning. To begin with, SODA limits the maximum size of messages to some (relatively large) network-dependent constant. In all likelihood, the limit could be respected by LYNX without packetizing messages and without seriously inconveniencing the programmer. After all, most language implementations place limits on all sorts of things: the length of variable names, the maximum depth of procedure nesting, the size of the run-time stack, and so forth.

A much stricter (and again unspecified) limit applies to the out-of-band information for *request* and *accept*. If all the self-descriptive information included in messages under Charlotte were to be provided out-of-band, a minimum of about 48 bits would be needed. With fewer bits available, some information would have to be included in the messages themselves, as in Charlotte.

The most serious problem with SODA involves a third unspecified constant: the permissible number of outstanding requests between a given pair of processes. The implementation described in the previous section would work easily if the limit were large enough to accommodate three requests for every link between the processes (a LYNX-request *put*, a LYNX-reply *put*, and a status *signal*). Since reply messages are always wanted (or can at least be discarded if unwanted), the implementation could make do with two outstanding requests per link and a single extra for replies. Too small a limit on outstanding requests would leave the possibility of deadlock when many links connect the same pair of processes. In practice, a limit of a half a dozen or so is unlikely to be exceeded (it implies an improbable concentration of simultaneously-active resources in a single process), but there is no way to reflect the limit to the user in a semantically-meaningful way. Correctness would start to depend on global characteristics of the process-interconnection graph.

None of these problems is a serious condemnation of the SODA design. At most, SODA would need only minor modifications to support a considerably simpler LYNX implementation than does Charlotte. There are at least three important lessons to be gained from this fact.

Lesson one: Hints can be better than absolutes.

> The maintenance of absolute, up-to-date, consistent, distributed information can be more trouble than it is worth. It may be considerably easier to rely on a system of hints, so long as they usually work, and so long as we can tell when they fail.
>
> The Charlotte kernel admits that a link end has been moved only when all three parties agree. The protocol for obtaining such agreement was a major source of problems in the kernel, particularly in the presence of failures and simultaneously-moving ends [7]. The implementation of links *on top of* SODA was comparatively easy.

Lesson two: Screening belongs in the application layer.

> Every reliable protocol needs top-level acknowledgments. A distributed

operating system can attempt to circumvent this rule by allowing a user program to describe *in advance* the sorts of messages it would be willing to acknowledge if they arrived. The kernel can then issue acknowledgments on the user's behalf. The shortcut only works if failures do not occur between the user and the kernel, and if the descriptive facilities in the kernel interface are sufficiently rich to specify precisely which messages are wanted. For implementing LYNX, the descriptive mechanisms of Charlotte were simply not rich enough.

SODA provides a very general mechanism for screening messages. Instead of asking the user to *describe* its screening function, SODA allows it to provide that function itself. In effect, it replaces a static description of desired messages with a formal subroutine that can be called when a message arrives.

Lesson three: Simple primitives are best.

From the point of view of the language implementor, the "ideal operating system" probably lies at one of two extremes: it either provides everything the language needs, or else provides almost nothing, but in a flexible and efficient form. A kernel that provides some of what the language needs, but not all, is likely to be both awkward and slow: awkward because it has sacrificed the flexibility of the more primitive system, slow because it has sacrificed its simplicity. Clearly, Charlotte could be modified to support all that LYNX requires. The changes, however, would not be trivial. Moreover, they would probably make Charlotte significantly larger and slower, and would undoubtedly leave out something that some other language would want. The beauty of SODA is that it provides mechanisms flexible enough to support a wide range of programming languages and styles.

Among the languages of chapter 1, existing implementations have all assumed a homogeneous environment. Implementors have felt free to adopt the first extreme, addressing themselves to the needs of a single language and often eliminating any real distinction between the operating system and the run-time support for the language itself. Such an approach will prove inadequate for general-purpose computing, when a machine like a multicomputer must be shared by dissimilar users. For such an environment, the kernel interface will need to be relatively primitive. The fact that LYNX can be implemented easily on something as simple as SODA speaks well of its appropriateness for writing general-purpose servers.

6. Measurements

6.1. Size

Object files produced by the Charlotte LYNX compiler tend to be about 50% larger than object files for comparable C programs. The difference can be attributed to a number of sources: default exception handlers, descriptive information for entries and messages, initialization, management of the environment tree, and run-time checks on subranges, sets, case statements, and function returns. In addition to the increase in basic code size, every LYNX program is linked to a substantial amount of run-time support code: the dispatcher, the communication routines, and code to manage exceptions and threads.

The run-time support consists of 23.7K bytes of object code. Of this total, 3.0K supports sets and run-time checks. It can legitimately be regarded as correcting deficiencies in C, rather than supporting the features of LYNX. The remaining 20.7K can be attributed to the following goals:

support for multiple threads of control	19%	3.9K
basic communication	29%	5.9K
multiple-link transfers	30%	6.1K
exception bookkeeping	11%	2.2K
exception handlers	12%	2.6K

The support for multiple-link transfers is divided more or less equally between finding links buried in data structures and packetizing messages. Somewhat less than half of the support for basic communication is devoted to forbid, allow, and retry messages, and to undersize buffers. I would expect the run-time support for SODA to be about 4K smaller than that for Charlotte. Both might be reduced further by careful programming.

6.2. Threads of Control

With a single thread of control, the following loop executes in just over 8.5 seconds:

```
foreach i in [1..100000] do
   await true;
end;
```

A loop with a call to an empty subroutine takes 3.8 seconds.

```
procedure null;
begin end null;

foreach i in [1..100000] do
   null;
end;
```

The loop overhead itself takes 0.6 seconds.


```
foreach i in [1..100000] do
   -- nothing
end;
```

By implication, a context switch between threads of control requires a minimum of 85 − 6 = 79 microseconds, or approximately two and a half times as long as a call to an empty subroutine with no arguments (38 − 6 = 32 microseconds).

Since the Boolean conditions in await statements can refer to arbitrary variables, the context of a thread must be restored before a condition can be checked. If there are many threads blocked at await statements, and if their Boolean conditions are relatively complicated (and therefore take some time to evaluate), it may take considerably longer than 79 microseconds to switch to a new ready task. Moreover, as pointed out in section 3.1, the run-time support routines must cycle through all tasks blocked at await statements before waiting for each external event.

Only extensive experience with LYNX will reveal whether the repeated evaluation of awaited conditions constitutes an unacceptable burden. It does not appear to be unacceptable in the applications written to date, mainly because await statements are infrequently used. Most threads block for communication, not local conditions.

If await statements should prove to be a burden, several steps could be taken:

- A more intelligent compiler could notice when threads are waiting on a simple Boolean variable, could keep those threads on a single queue, and could check the variable only once, without changing contexts.
- The language could be extended to allow Boolean expressions to be associated with named **condition** variables, as proposed by Kessels [71] for use

in monitors. Run-time routines could then check even complicated conditions exactly once when the current thread blocks, again without changing contexts.

- The language could be extended to include semaphores or signals, requiring threads to unblock each other explicitly.

6.3. Communication

The facilities of LYNX are not without cost. Even the simplest remote request must gather and scatter parameters, manage queues of sending and receiving threads, establish default exception handlers, enforce flow control, check operation names and types, guard against buffer overflow, look for enclosed links, and make sure the links are valid.

The following table summarizes timing information for two simple operations. The first half of the table gives times for a remote operation with no request or reply parameters. The second half gives times for an operation with 1000 bytes of data transfer in each direction. In each row, times for LYNX programs are compared against times for C programs that execute the same system calls in the same order but without the checks described above. The figures were obtained by timing a pair of processes in tight 1000-iteration loops, one with an embedded connect statement, the other with an accept statement.[13] The 6 microsecond loop overhead is insignificant.

[13] Tests were carried out over a period of several days under a variety of network loads. Tests that demonstrated high variance were repeated more often than those that seemed more stable. No test was repeated fewer than five times. The ± figures give the most significant digit of the standard deviation.

	explicit receipt	
	LYNX	**C**
Empty request, reply		
intra-machine	45.6 ± 0.1 ms	39.4 ± 0.3 ms
inter-machine	56.9 ± 0.8 ms	54.7 ± 0.7 ms
1K request, reply		
intra-machine	49.7 ± 0.1 ms	41.1 ± 0.1 ms
inter-machine	65.1 ± 0.1 ms	60.1 ± 0.1 ms

The gap between LYNX and C programs is smaller across machines because parts of the run-time support can execute in parallel. The gap between the times in the first and second halves of the table is due in part to the copying of buffers. A timing test similar to the one for context switches reveals that copying a 1000 byte buffer requires approximately 360 microseconds. The LYNX program requires 4 such copies for gathering and scattering, for a total of 1.4 milliseconds. The rest of the gap is consumed by the Charlotte kernel.

Similar figures have been obtained for implicit message receipt:

	implicit receipt	
	LYNX	**C**
Empty request, reply		
intra-machine	46.8 ± 0.1 ms	39.5 ± 0.3 ms
inter-machine	60.2 ± 0.1 ms	55.2 ± 0.2 ms
1K request, reply		
intra-machine	50.3 ± 0.1 ms	41.1 ± 0.2 ms
inter-machine	69.5 ± 0.1 ms	63.3 ± 0.3 ms

Differences from the first table are due to two factors. First, the LYNX server in the second table incurs overhead for the creation and destruction of a separate thread of control for each request. Second, the speed of the kernel itself is affected by the order in which system calls are made. With implicit receipt, the

run-time system posts a new *Receive* as soon as the old one completes.

server with explicit receipt	server with implicit receipt
Receive	Receive
Wait (receive)	Wait (receive)
Send	Receive
Wait (send)	Send
Receive	Wait (send)
Wait (receive)	Wait (receive)
Send	Receive
Wait (send)	Send
Receive	Wait (send)
...	...

Because of Charlotte's sensitivity to the ordering of events, the figures above are suggestive, not definitive. The speed of messages in practice will depend not only on the actions of the sender and receiver, but equally well on all events that are noticed by the kernel.

6.4. Predicted Values for SODA

The SODA LYNX implementation, as described in section 5.3, would be considerably simpler than the one for Charlotte. Most of the difference, however, would be seen only in unusual cases: the transfer of messages with multiple enclosed links, the receipt of unwanted requests. Typical message traffic would require about as much run-time support as it currently does in Charlotte. It might, however, require considerably less kernel support.

It is difficult to compare message transmission times in Charlotte and SODA. Charlotte has a considerable hardware advantage: the only implementation of SODA ran on a collection of PDP-11/23's with a 1-Mbit/second CSMA bus. SODA, however, probably has a software advantage: it was much simpler and easier to implement. The Charlotte group made a deliberate decision to sacrifice efficiency in order to keep the project manageable.

With these reservations in mind, it appears reasonable to expect considerably better performance from SODA. Experimental figures reveal that for small messages SODA was three times as fast as Charlotte.[14] Much of the difference can be attributed to the *lack* of features in the kernel. By providing a simpler interface, SODA avoids duplicating functions that are provided at a higher level.

To a large extent, the differences between Charlotte and SODA can be cast in the context of a more general class of **end-to-end arguments** [97]. End-to-end arguments provide a rationale for simplifying the lower levels of a layered software system. Among other things, they question the wisdom of providing too many functions in levels that are shared by several applications. Any facility that is not used by a given application will extract a performance penalty that could be avoided by moving it up into the higher levels that use it.

One of the easiest targets for end-to-end arguments is the detection of errors in communication protocols. A lower protocol level can only eliminate errors that can be described in the context of its interface to the level above. Overall reliability must be ensured at the application level. Since end-to-end checks generally catch *all* errors, low-level checks are redundant. They are justified only if errors occur frequently enough to make early detection essential.

---

[14] The difference is less dramatic for larger messages; SODA's slow network extracted a heavy toll. The figures break even somewhere between 1K and 2K bytes.

The run-time system for LYNX never passes Charlotte an invalid link end. It never specifies an impossible buffer address or length. It never tries to send on a moving end or enclose an end on itself. To a certain extent it provides its own top-level acknowledgments, in the form of goahead, retry, and forbid messages, and in the confirmation of operation names and types implied by a reply message. It would provide additional acknowledgments for reply messages (section 4.3, paragraph 2) if they were not so expensive. For the users of LYNX, Charlotte wastes time by checking these things itself.

Conclusion

This dissertation makes at least five important contributions to computer science.

(1) It enumerates the language needs of multicomputer systems programs.

(2) It presents a framework for the discussion of distributed languages and a survey of previous proposals.

(3) It develops a new language ideally suited to meeting the needs in (1).

(4) It demonstrates the feasibility of implementing that language (and by implication languages in general) on a distributed operating system.

(5) It derives useful insights into the nature of the language/operating system interface.

In comparison to a sequential language that performs communication through library routines or through direct calls to operating system primitives, LYNX supports

- direct use of program variables in communication statements
- secure type checking
- thorough error checking, with exception handlers outside the normal flow of control
- automatic management of concurrent conversations

In comparison to previous distributed languages, LYNX obtains these benefits without sacrificing the flexibility needed for loosely-coupled applications. LYNX supports

- dynamic binding of links to processes
- dynamic binding of types to links
- abstraction of distributed resources
- protection from errors in other processes

In addition, LYNX reflects the structure of an underlying multicomputer by differentiating between processes, which execute in parallel and pass messages, and threads of control, which share memory and execute in mutual exclusion.

The languages of chapter 1 were designed primarily to support communication between pieces of a single distributed program. Even for this limited domain, LYNX offers some advantages over most previous proposals. By providing both explicit and implicit receipt, LYNX admits a wide range of communication styles. By allowing dynamic binding of links to entry procedures, LYNX provides access control for such applications as the readers/writers problem. By integrating implicit receipt with the creation of threads, LYNX supports communication between processes and management of context within processes with an economy of syntax.

Support for tightly-coupled programs, however, is not central to the thesis. The real significance of the work at hand lies in problems unaddressed by previous research. LYNX is not another language in the mold of chapter 1. It meets the needs of loosely-coupled applications for which other languages were never intended. I feel no need to claim that LYNX is better than other languages for general-purpose use, only that it is more appropriate for writing servers on a multicomputer operating system.

Since the arguments for LYNX rest on practical (as opposed to theoretical) concerns, they are strengthened considerably by the existence of a compiler and a working run-time system. The implementation for Charlotte, as described in chapter 4, is important in several respects. It permitted the construction of usable programs. It verified that the language could be implemented efficiently, with a reasonable amount of effort. It spurred the development of novel techniques for checking types and moving links. In concert with the paper design for SODA, it produced the lessons of section 5.3 (hints can be better than absolutes; screening belongs in the application layer; simple primitives are best). Finally, it resulted in a product of continuing value to the larger Charlotte project.

Directions for Future Research

The material in this dissertation suggests several avenues for future work. The most obvious of these would explore extensions and improvements to LYNX. Some extensions could be motivated by efforts to adapt the language to additional problem domains. Substantial changes might be needed, for example, if LYNX were to be used on a multiprocessor architecture, where memory could be shared by more than one CPU. Other extensions might prove useful even in the language's original domain. Possibilities include:

- A *cobegin* construct for dividing a thread into subthreads

  Such a construct would, for example, allow a thread to request operations on two different links when order is unimportant. As currently defined, LYNX requires the thread to specify an arbitrary order, or else create subthreads through calls to entries that are separated lexically from the principal flow of control.

- A mechanism for forwarding requests

  In some algorithms a server passes a request on to a peer, waits for a reply, and passes the reply back to the client. Communication could be reduced if the second server replied to the client directly. It is not immediately obvious how such a facility would work in LYNX, where all communication is constrained to flow through links.

- "Exception handlers" for bindings

  A link that is bound to entries but is not in use by any active thread can be destroyed at either end. LYNX specifies that the bindings are broken, but provides no "hook" for programmer-defined recovery. The language could allow a link to be bound to a **cleanup entry** that would be executed automatically when the link was destroyed.

- Asynchronous receipt

  As currently defined, LYNX provides no mechanism for coping with asynchronous external events. Both incoming and outgoing messages go unnoticed until all threads in a process are blocked. Real-time device control cannot be programmed in LYNX, nor can any algorithm in which incoming messages must *interrupt* the execution of lower-priority "background" computation [42, 46]. To support such algorithms, LYNX would need considerably more elaborate facilities than it currently provides for synchronizing threads.

Beyond mere changes to the language, work on LYNX points to several related subjects. Efficiency is one of these, particularly the relationship of efficiency to *layers* in a communication protocol. Previous work on high-speed mechanisms for interprocess communication [77, 90, 103] suggests that layers extract an enormous price in communication overhead. Nevertheless, layers are certainly useful. Their modular structure makes them easier than integrated systems to build, debug, maintain, and understand. They also promote flexibility by allowing different upper layers to run simultaneously on the same underlying system. Research to date has produced neither a really fast layered protocol nor a convincing explanation of why such a protocol cannot exist.

Assuming that layers will continue to exist, at least to the extent that languages will be built on top of distributed operating systems, it is very much worthwhile to investigate the interface between the layers. The results in chapter 4 suggest that the interface defined by the Charlotte kernel is inappropriate: too low-level to be used conveniently "as is," yet too inflexible to support a straightforward implementation of LYNX. The search for an "ideal" interface would benefit substantially from formal criteria for evaluating particular proposals.

Finally, there is at least some reason to be suspect of any programming language that places "too many" parameters outside the control of the individual programmer. Niklaus Wirth, in his 1984 Turing Award Lecture, proposed that "Systems programming requires an efficient compiler generating efficient code that operates without a fixed, hidden, and large so-called run-time package." [119] LYNX does not meet this standard by any stretch of the imagination. Its run-time package *is* fixed, hidden, and large, for the simple reason that it cannot itself be written in LYNX. The package relies on knowledge of the compiler's storage-allocation strategy. It also uses symbol-table data generated by the compiler but invisible to the user. If the necessary information were made an explicit part of the language, much of the run-time package could be moved into library routines. Pieces that proved inappropriate for particular applications could be quickly and easily replaced. The modified language might be considerably more flexible than the current version. It might also be more amenable to formal analysis (though its library routines might not!).

The design of LYNX was very much an exercise in practical problem-solving. As such, it must be judged on the basis of the solutions it provides. Only long-term experience can support a final verdict. New problems will

undoubtedly arise and will in turn provide the impetus for additional research. At present, however, the evidence suggests that LYNX is a success.

Appendix

LYNX Reference Manual

Caveat: this reference manual is not a formal document. It describes the Charlotte/VAX implementation of LYNX at a level of detail suitable for programmers.

1. Lexical Conventions

A LYNX program is a sequence of characters from the Ascii character set. Characters are scanned by the compiler from left to right and are grouped together in tokens. Some tokens are valid prefixes of longer tokens. In such cases, the compiler finds tokens of maximal length. Tokens can be separated by white space (spaces, tabs (\t), and newlines (\n)). White space has no other meaning.

Many tokens are simple symbols:

```
,    ;    :    .    ..   |
(    )    [    ]    {    }    <<   >>
:=   +    -    *    /    ~    ~>
<    <=   >=   >    =    <>
```

Others are more complicated. All can be defined by regular expressions.

In the following, italics are used for intermediate definitions. Parentheses are used for grouping. Vertical bars are used for alternation. Other adjacent symbols are meant to be concatenated. The function NOT indicates complementation with respect to the Ascii character set.

Comments in LYNX begin with '- - ' and extend through end-of-line.

```
COMMENT =
    - - ( NOT ( \n ) ) *
```

Comments are treated like white space.

Numeric constants can be expressed in octal, decimal, or hexadecimal.

```
NUMBER =
    0 ( 0 | octdigit ) * |
    decdigit ( 0 | decdigit ) * |
    # ( 0 | hexdigit ) *
where
    octdigit = '1'..'7'
    decdigit = '1'..'9'
    hexdigit = '1'..'9', 'A'..'F', 'a'..'f'
```

Character and string constants are delimited by single and double quotes, respectively. Non-printing characters may be indicated by the single-letter backslash-escapes of C (\b, \n, \r, \t), or by numbers (as defined above) delimited by a *pair* of backslashes (as in \#7f\ for the delete character). Single quotes in character constants and double quotes in string constants are indicated by \' and \", respectively. Backslashes are indicated by \\. Backslashes not accounted for by any of the preceding rules are ignored.[15]

[15] These conventions agree with C except in the form of numeric escapes.


```
CHARCONST =
    ' (
        NOT ( ' , \ , \n , nonprint ) |
        \ NOT ( # , 0 , decdigit , \n , nonprint ) |
        \ number \
    ) '
STRINGCONST =
    " (
        NOT ( " , \ , \n , nonprint ) |
        \ NOT ( # , 0 , decdigit , \n , nonprint ) |
        \ number \
    ) * "
where
    nonprint indicates the Ascii characters with codes 1..8, 11..31, and 127
    decdigit is as above
    number is as defined for the token ''number''
```

Keywords are:

```
ACCEPT      AND         ARRAY       AWAIT
BEGIN       BIND        CALL        CASE
CONNECT     CONST       DO          ELSE
ELSIF       END         ENTRY       EXCEPTION
EXIT        EXPORT      EXTERNAL    FOREACH
FORWARD     FROM        FUNCTION    IF
IMPORT      IN          LOOP        MOD
MODULE      NOT         OF          ON
OR          OTHERWISE   PROCEDURE   RAISE
READ        RECEIVE     RECORD      REMOTE
REPEAT      REPLY       RETURN      REVERSE
SEND        SET         THEN        TO
TYPE        UNBIND      UNTIL       VAR
WHEN        WHILE       WITH        WRITE
```

After excluding keywords, identifiers are strings of letters, digits, and underscores that do not begin with a digit and do not end with an underscore. Case is not significant in identifiers, except when significance is imposed from outside by associating names in the language with external objects.

```
IDENTIFIER =
    letter | (
        ( letter | _ )
        ( letter | _ , digit ) *
        ( letter | digit )
    )
where
    letter = 'A'..'Z', 'a'..'z'
    digit = '0'..'9'
```

2. Types

A type is a set of values and a mapping from those values to representations in memory. Types are useful for restricting the values that can be used in various contexts. Several types are pre-defined. Others are created by type constructors.

```
type        ::= IDENTIFIER
            ::= enum_type
            ::= subr_type
            ::= array_type
            ::= record_type
            ::= set_type
```

2.1. Pre-defined Types

integer

consists of as many distinct values as can be represented in a single word. Its values lie in a contiguous range centered approximately at the origin (−2147483648 through 2147483647 on the VAX).

char

consists of the Ascii characters. Char variables occupy one byte on the VAX.

Boolean

consists of the truth values. *True* and *false* are pre-defined constants of type Boolean. Boolean variables occupy one byte on the VAX.

link

consists of references to the ends of communication channels. Link values are created at run time. A given end of a given link is accessible to only one process at a time. Links are discussed in detail below.

Nolink is a pre-defined constant of type link. The value nolink can be assigned into or compared against the contents of a link variable, but is usable for nothing else. Link variables occupy two bytes on the VAX.

2.2. Enumerations

The values of an enumeration type have a one-one correspondence with the first few non-negative integers.

```
enum_type       ::= ( ident_list )
ident_list      ::= IDENTIFIER id_list_tail
id_list_tail    ::= , ident_list
                ::=
```

The identifiers in the list name the values of the type. Enumeration variables occupy four bytes on the VAX.

2.3. Subranges

A type can be declared to be a subrange of any existing **scalar** type. The existing type is called the **parent type** of the subrange. Scalar types are integers, chars, Booleans, enumerations, and subranges.

```
subr_type       ::= [ expr .. expr ]
```

Subrange variables occupy one, two, or four bytes on the VAX, depending on whether or not their bounds fall in the ranges −128..127, −32768..32767, or −2147483648..2147483647, respectively.

2.4. Array Types

The values of an array type are ordered lists of values of the array's **element type**. The length of each list is the number of distinct values of the array's **index type**.

```
array_type      ::= ARRAY type OF type
```

The type that follows the word ARRAY is the index type. The second type is the element type. The index type must be scalar.

A variable of an array type thus consists of many smaller variables, called the **elements** of the array. The element variables have names: if *expr* is an expression whose type is the index type of array "foo" and whose value is *n*, then "foo [*expr*]" is a name for the *n*th element of foo.

The elements of an array are stored in consecutive locations in memory. The location of the first element is the same as the location of the array. There is no special syntax for multi-dimensional arrays. The programmer can of course declare arrays of arrays and access their elements as "name [row] [column]."

2.5. Record Types

A record is a list of named *fields*. The values of a record type are lists of values of the types of the fields.


```
record_type     ::= RECORD field_list_opt END
field_list_opt  ::= field field_list_opt
                ::=
field           ::= ident_list : type ;
                ::= CASE IDENTIFIER : type OF vnt_list_opt END ;
                ::= ;
vnt_list_opt    ::= variant vnt_list_opt
                ::=
variant         ::= { component_list } field_list_opt
component_list  ::= component comp_list_tail
comp_list_tail  ::= , component comp_list_tail
                ::=
component       ::= expr component_tail
component_tail  ::= .. expr
                ::=
```

A variable of a record type thus consists of a collection of smaller variables, one for each field of the record. Each of these smaller variables has a name. The name is created by appending a period and the name of the field to the name of the record variable.

The word CASE introduces a variant portion of a record. All variants have the same location. Only one variant is **valid** at a time.

The identifier following the word CASE is the name of a special field called the **tag** of the variant portion of the record. The tag must have a scalar type. It determines which variant is valid at a particular point in time. The component list of each variant lists the values of the tag for which the variant is valid. The lists must be disjoint. Their component expressions must have values that can be determined at compile time. They cannot involve function calls.

On the VAX, records have the same representation in memory as C structs and unions.

2.6. Set Types

The values of a set type are unordered sets of values of the set's **component type**.

```
set_type        ::= SET OF type
```

The component type must be scalar or link.

On the VAX, every set variable occupies 16 bytes. The component type of a set (if other than *link*), must have no more than 128 elements. For sets of subranges of integers, the subrange bounds must lie between 0 and 127, inclusive.

3. Declarations

Identifiers denote types, constants, variables, exceptions, exception classes, subroutines, and entries. Several identifiers are pre-defined; all others must be declared by the programmer. Identifiers exported from a module (sections 3.7 and 4) appear in the export list before they are declared. All other identifiers must be declared before they are used.

```
dec_pt          ::= declaration dec_pt
                ::=
declaration     ::= CONST const_dec const_dec_tail
                ::= TYPE type_dec type_dec_tail
                ::= VAR var_dec var_dec_tail
                ::= EXCEPTION ident_list ;
                ::= subroutine ;
                ::= entry ;
                ::= module ;
```

Declaration sections can appear in any order, an arbitrary number of times.

3.1. Types

A type may have any number of names. There are four built-in types. They all have names. Each lexical occurrence of a type constructor introduces a new type. A type declaration introduces a new name for the type on its right hand side. A constructed type that appears on the right-hand side of a variable declaration has no name. Once declared, the name of a type can be used anywhere a type constructor could be used, but without introducing a new type.

```
type_dec        ::= IDENTIFIER = type ;
                ::= ;
type_dec_tail   ::= type_dec type_dec_tail
                ::=
```

3.2. Constants

Constant declarations introduce names for string constants or for values of pre-defined scalar types.

```
const_dec       ::= IDENTIFIER = expr ;
                ::= ;
const_dec_tail  ::= const_dec const_dec_tail
                ::=
```

The expression must have a value that can be determined at compile time. It cannot involve function calls. If the expression is a string constant, then the declared constant has the new and nameless type "ARRAY [0..n] of char," where n is the number of characters in the string. Byte n is a null (Ascii 0).

3.3. Variables

Variable declarations reserve memory locations and introduce names for those locations.

```
var_dec          ::= ident_list : type ;
                 ::= ;
var_dec_tail     ::= var_dec var_dec_tail
                 ::=
```

The name of a variable refers either to the location of the variable or to the value stored at that location (its **contents**), depending on context. The type of a variable restricts the values that can be stored at its location. It is a programming error to refer to the contents of a variable before storing a value at its location.

3.4. Exceptions

Every link value has several exceptions associated with it. Names for these exceptions consist of an expression of type link (with the appropriate value) followed by the name of a built-in **exception class**.

Additional exceptions are introduced by exception declarations. Each identifier in the identifier list of an exception declaration is the (only) name for a new exception.

Programmer-defined exceptions have different semantics from the exceptions associated with links (see section 7.2). Both kinds of exceptions are used only in when clauses (section 6.11) and raise statements (section 6.12).

3.5. Subroutines

Subroutines are parameterized sequences of statements.


```
subroutine       ::= PROCEDURE IDENTIFIER arg_list_opt ; body
                 ::= FUNCTION IDENTIFIER arg_list_opt
                     fun_type_opt ; body
arg_list_opt     ::= ( mode formal more_m_formals )
                 ::=
more_m_formals   ::= ; mode formal more_m_formals
                 ::=
mode             ::= VAR
                 ::= CONST
                 ::=
formal           ::= ident_list : IDENTIFIER
fun_type_opt     ::= : IDENTIFIER
                 ::=
body             ::= dec_pt compound_stmt IDENTIFIER
                 ::= FORWARD
                 ::= EXTERNAL
```

The name of the subroutine follows the keyword PROCEDURE or FUNCTION. The identifier at the end of a non-trivial subroutine body must match the name of the subroutine.

The argument list specifies **formal parameters** for the subroutine, together with the modes and type names of those parameters. Within the compound statement of a subroutine body, parameters may be used as if they were variables. VAR parameters are passed by reference. Plain parameters are passed by value. The contents of CONST parameters cannot be modified. CONST parameters are passed by reference on the VAX.

Functions yield a value whose type name follows the argument list. Procedures do not yield a value.

A FORWARD subroutine body indicates that the subroutine will be declared again later in the same scope, with a non-trivial body. The second declaration must omit the argument list and function type.

An EXTERNAL subroutine body indicates that the subroutine is external to LYNX and must be found by the linker. Case is significant in the names of external subroutines.

3.6. Entries

An entry resembles a procedure.

```
entry           ::= ENTRY IDENTIFIER in_args_opt out_types_opt ;
                    body
in_args_opt     ::= ( formal formal_tail )
                ::=
formal_tail     ::= ; formal formal_tail
                ::=
out_types_opt   ::= : ident_list
                ::=
```

The name of the entry follows the keyword ENTRY. As with subroutines, the identifier at the end of a non-trivial body must match the name of the entry.

An entry cannot be declared EXTERNAL, but it can be declared REMOTE.

```
body            ::= REMOTE
```

A REMOTE body indicates that the entry may de declared again (minus in arguments and out types, but with non-trivial body) in the same scope. Unlike FORWARD, REMOTE does not *require* the later declaration.

The in arguments and out types of an entry are templates for the request and reply messages of a remote operation. Within the statements of the body of the entry, the in arguments can be used as if they were variables. Through the use of the **bind** statement (section 6.9), the programmer can arrange for an entry to be executed in response to incoming requests.

3.7. Modules

Modules are an encapsulation mechanism for structuring programs and for limiting the scope of identifiers.

```
module          ::= MODULE IDENTIFIER ; import_pt export_pt
                    dec_pt cpd_stmt_opt IDENTIFIER
import_pt       ::= IMPORT ident_list ;
                ::=
export_pt       ::= EXPORT ident_list ;
                ::=
cpd_stmt_opt    ::= compound_stmt
                ::= END
```

The compound statement of a module, if it has one, is called the module's **initialization code**. For consistency with the terms for subroutines and entries, it is occasionally called the module's **body** as well. The purpose of import and export lists is explained below.

4. Scope

Declaration sections appear near the beginning of every **block**. Blocks are subroutines, entries, and modules.

Declarations introduce **meanings** for identifiers. Identifiers can have different meanings at different places in a program. The portion of a program in which a particular meaning holds is called that meaning's **scope**. The scope of a meaning extends from the declaration of its identifier to the end of the block in which that declaration appears, with three exceptions:

(1) If a nested block contains a declaration of the same identifier, or if a with statement or labeled statement introduces a new meaning for the identifier (see sections 6.5.4, 6.6, and 6.8), then the scope of the outer meaning

does not include the scope of the inner meaning.

(2) A meaning does not extend into any nested module unless its identifier is explicitly **imported**.

(3) If a module explicitly **exports** an identifier, then the meaning of that identifier extends from its declaration inside the module to the end of the *enclosing* block (subject to exceptions (1) and (2)).

Identifiers can be imported or exported repeatedly in a nested chain of modules.

For the purpose of defining scopes, the formal parameters of subroutines and entries are considered to be part of the declaration section immediately following their argument list. They are not visible in as large a scope as is the name of their subroutine or entry.

Two record types visible at the same point in a program can have fields with the same name. Otherwise, declarations of the same identifier must have disjoint scopes. In particular, simultaneously visible enumeration types cannot have values with the same name.

The **environment** of a particular thread of control at run time is a mapping from names to their current meanings. New meanings appear whenever control enters the initialization code of a module or the body of a subroutine or entry. The affected identifiers are those declared in the immediately preceding declaration section. For a subroutine or entry, the meanings disappear with the completion of the body of the block. For a module, the meanings disappear with the completion of the closest enclosing subroutine or entry. They may not be visible outside the module, unless they are exported. For any particular thread, the appearance and disappearance of meanings occurs in LIFO order. (The same is not true of a process as a whole, as discussed in section 7.)

5. Expressions

An expression **evaluates** to a value at run time. Every expression has a type. Expressions are composed of atoms, parentheses, function calls, and operators.

```
expr            ::= term expr_tail
expr_tail       ::= rel_op term expr_tail
                ::=
term            ::= factor term_tail
term_tail       ::= other_op factor term_tail
                ::=
factor          ::= NOT factor
                ::= - factor
                ::= constant
                ::= set
                ::= ( expr )
                ::= selector sel_fac_tail
selector        ::= IDENTIFIER selector_tail
selector_tail   ::= . IDENTIFIER selector_tail
                ::=
```

5.1. Atoms

An atom is an explicit constant, or the name of a constant or variable.

```
constant        ::= NUMBER
                ::= CHARCONST
                ::= STRINGCONST
sel_fac_tail    ::= changeover
                ::=
changeover      ::= [ expr ] designator_tail
                ::= : IDENTIFIER designator_tail
designator_tail ::= . IDENTIFIER designator_tail
                ::= changeover
                ::=
```

A number or character constant is an expression with an obvious value and type. A string constant has the new and nameless type "ARRAY [0..n] of char," where n is the number of characters in the string.

The name of a constant or variable is an expression whose value is the value of the constant or the contents of the variable and whose type is the type of the constant or variable. Within a name, a period indicates selection of a field of a record. Brackets indicate selection of an element of an array. A colon indicates a type cast.

Type casts are allowed only on variables. A variable name followed by a type cast is the name of an imaginary variable whose type is specified by the cast, whose location is the same as that of the original variable, and whose value is determined by interpreting the data at that location. That value may be garbage.

5.2. Set Expressions

A set expression evaluates to a value of type "SET OF *component_type*," where *component_type* is a subrange whose bounds are the lowest and highest possible values of any of the component expressions or ranges. The set type is new and nameless. It is **provisional** in the sense that it may be **coerced** to another type if context requires it.

```
set             ::= { comp_list_opt }
comp_list_opt   ::= component_list
                ::=
```

The value of the set contains the value of each component expression and all values in each component range.

5.3. Function Calls

The type of a function call is specified in the declaration of the function. The value is obtained by **invoking** the function at run time.

```
sel_fac_tail    ::= ( expr_list )
expr_list       ::= expr expr_list_tail
expr_list_tail  ::= , expr_list
                ::=
```

The expressions in the argument list are called **actual parameters**. They must agree in order and number with the formal parameters of the function. Their types must be **compatible** with the types of the formals. Type compatibility is discussed under assignment statements (section 6.1). A function call with no parameters looks like an atom.

The values of the actual parameters are used as initial values for the formal parameters of the function. Actual parameters corresponding to VAR or CONST formal parameters must be variables. The contents of actual parameters corresponding to VAR formal parameters may be changed by invoking the function. The contents of actual parameters corresponding to value or CONST formal parameters are not changed.

5.4. Operators

All operators are pre-defined. They are represented by the following tokens:

```
+      -      *      /      ~      ~>
<      <=     >=     >      =      <>
NOT    AND    OR     IN     MOD
```

NOT is a **unary** operator. It has one **operand**, the expression to its right. The

minus sign (−) can also be a unary operator, if there is no expression to its immediate left. Otherwise, it is a **binary** operator. Binary operators have two operands: the expressions to their left and right. The rest of the operators in the above list are binary.

5.4.1. Operator Precedence

In the absence of parentheses, operands and operators are grouped together according to the following levels of precedence.

Loosest grouping

OR
AND
< <= >= > = <>
+ − *(binary)*
* MOD /
NOT − *(unary)*

Tightest grouping

Operators of equal precedence associate from left to right.

5.4.2. Operator Semantics

For the purposes of this section, define the **base** of any type except a subrange to be the type itself. Define the base of a subrange to be the base of the subrange's parent type.

NOT

is a unary operator whose operand must have base type Boolean. "NOT *expr*" is an expression of type Boolean whose value is the negation of the value of *expr*.

AND and OR

are binary operators whose operands must have base type Boolean. "*expr1* AND *expr2*" and "*expr1* OR *expr2*" are expressions of type Boolean whose values are the logical *and* and *or*, respectively, of the values of their operands.

(unary) −

is an operator whose operand must have base type integer. "− *expr*" is an expression of type integer whose value is the additive inverse of the value of *expr*.

+, −, and *

are binary operators whose operands must be sets, or else of base type integer. If *expr1* and *expr2* are of base type integer, then "*expr1* + *expr2*," "*expr1* − *expr2*," and "*expr1* * *expr2*" are expressions of type integer whose values are the sum, difference, and product, respectively, of the values of their operands. The VAX implementation performs these operations in two's complement arithmetic with no checks for overflow.

If *expr1* and *expr2* are sets, then "*expr1* + *expr2*," "*expr1* − *expr2*," and "*expr1* * *expr2*" are expressions whose values are the union, difference, and intersection, respectively, of the values of the operands. If neither operand has a provisional type, then the types must be the same, and the type of the expression will be the same as well. If exactly one operand has a provisional type, then it is coerced to the type of the other operand, if possible. The coercion is not permitted if 1) the two operands have different component base types, or 2) the bounds of the component type of the provisional operand do not lie within the bounds of the component type of the

other operand. If both operands have provisional types, then the bases of their component types must be the same, and the expression has a new provisional type. The component type of the expression has the same base as the component types of the operands, and its bounds are the minimum and maximum of the bounds of the components of the operands.

/ and MOD

are binary operators whose operands must have base type integer. "*expr1* / *expr2*" and "*expr1* MOD *expr2*" are expressions of type integer whose values are the quotient and remainder, respectively, obtained in dividing *expr1* by *expr2*. The remainder has the same sign as the dividend (in this case *expr1*).

<, <=, >=, and >

are binary operators whose operands must either be sets or else have scalar base types. If the operands are sets, then the type rules described under "+, -, and *" apply. "*set1* op *set2*" is an expression of type Boolean whose value reflects the relationship between the two sets. In the order of the heading above, the operators determine whether *set1* is a proper subset, subset, superset, or proper superset of *set2*.

If the operands are scalars, then their base types must be the same, and "*expr1* op *expr2*" is an expression of type Boolean whose value indicates whether *expr1* is less than, less than or equal to, greater than, or greater than or equal to *expr2*.

= and <>

are binary operators whose operands must either be sets, be of type link, or have scalar base types. If the operands are sets, then the type rules

described under "+, -, and *" apply. If the operands are scalars, then their base types must be the same. In all cases, "*expr1* op *expr2*" is an expression of type Boolean whose value indicates whether *expr1* and *expr2* have the same value.

~ is a binary operator whose operands must have type link. "*expr1* ~ *expr2*" (read "*expr1* is similar to *expr2*") is an expression of type Boolean whose value indicates whether the values of *expr1* and *expr2* are references to opposite ends of the same link. Checking for similarity is not supported by the Charlotte implementation.

~> is a binary operator whose left operand must have type link and whose right operand must be the name of an entry. "*expr* ~> entryname" is an expression of type Boolean whose value indicates whether the link end referenced by *expr* is **bound** to entryname. (Bindings are discussed in section 6.9.)

IN is a binary operator whose right operand must be a set whose component base type is the same as the base type of the left operand. "*expr1* IN *expr2*" is an expression of type Boolean whose value indicates whether the value of the left operand is a component of the value of the right operand.

6. Statements

Statements accomplish the work of a program. They change the contents of variables, send messages, and produce output data on the basis of internal calculations, incoming messages, and input data.

```
stmt            ::= reply
                ::= other_stmt
other_stmt      ::= label_opt labeled_stmt
                ::= communication
                ::= io
                ::= bind_stmt
                ::= unbind_stmt
                ::= if_stmt
                ::= case_stmt
                ::= exit_stmt
                ::= with_stmt
                ::= return_stmt
                ::= await_stmt
                ::= raise_stmt
                ::= selector sel_stmt_tail
                ::=
labeled_stmt    ::= loop_stmt
                ::= compound_stmt
```

6.1. Assignment Statement

An assignment statement changes the contents of a variable.

```
sel_stmt_tail   ::= firstchange := expr
firstchange     ::= changeover
                ::=
```

The **left-hand side** of the assignment precedes the := sign. It must be the name of a variable. The type of the expression on the **right-hand side** must be **compatible** with the type of the left-hand side.

Every type is compatible with itself (compatibility is reflexive). A subrange and its parent type are compatible with each other. Two subranges are compatible with each other if their parent types are compatible and if their sets of values intersect. (Run time checks may be necessary to guarantee that assignments produce valid values for the left-hand side.) A string constant is compatible with any array whose elements have base type char. A long string may be truncated to fill

a small array. A short string may be extended with garbage to fill a large array. A provisional set type is compatible with any type it could be coerced to match (run time checks may again be necessary). Types not covered by these rules are not compatible.

6.2. Procedure Call

Like a function call, a procedure call provides a set of **actual parameters** to be used for the initial values of the formal parameters of the subroutine. Unlike a function, a procedure yields no value.

```
sel_stmt_tail   ::= e_arg_opt
e_arg_opt       ::= ( expr_list )
                ::=
```

Actual parameters must agree in order and number with the formal parameters of the procedure. Their types must be compatible with the types of the formals. Actual parameters corresponding to VAR or CONST formal parameters must be variables. The contents of actual parameters corresponding to VAR formal parameters may be changed by calling the procedure. The contents of actual parameters corresponding to value or CONST formal parameters are not changed.

6.3. If Statement

An if statement contains one or more lists of statements, at most one of which is executed. The choice between the lists is based on the values of one or more Boolean expressions.

```
if_stmt          ::= IF expr THEN stmt_list_opt elsif_list_opt else_opt
                     END
elsif_list_opt   ::= ELSIF expr THEN stmt_list_opt elsif_list_opt
                 ::=
else_opt         ::= ELSE stmt_list_opt
                 ::=
stmt_list_opt    ::= stmt ; stmt_list_opt
                 ::=
```

The first statement list is executed if the first Boolean is true, the second if the second Boolean is true, and so forth. The last list, if present, is executed if none of the Booleans are true.

6.4. Case Statement

Like an if statement, a case statement contains multiple lists of statements. It is intended for the commonly-occurring situation in which the choice between lists is based on the value of a single variable.

```
case_stmt        ::= CASE expr OF case_list_opt default_opt END
case_list_opt    ::= { component_list } stmt_list_opt case_list_opt
                 ::=
default_opt      ::= OTHERWISE stmt_list_opt
                 ::=
```

The expression following the word CASE must be a scalar. The beginning of each **arm** of the case statement has the same syntax as a set expression. The component lists must be disjoint. The expressions they contain must have values that can be determined at compile time. They cannot involve function calls.

Exactly one of the statement lists must be executed. If the value of the scalar expression is found in one of the component lists, then the immediately following statement list is executed. If the value is not found, then the statement list following the word OTHERWISE (if present) is executed instead. If the value is not found and the OTHERWISE clause is missing, then an error has occurred and execution must halt.

6.5. Loop Statements

Loop statements cause repetitive execution of a nested list of statements.

```
loop_stmt        ::= forever_loop
                 ::= while_loop
                 ::= repeat_loop
                 ::= foreach_loop
```

6.5.1. Forever Loop

Execution can only leave a forever loop by means of an exit statement, a return statement, or an exception.

```
forever_loop     ::= LOOP stmt_list_opt END
```

6.5.2. While Loop

The header of a while loop contains a Boolean expression.

```
while_loop       ::= WHILE expr DO stmt_list_opt END
```

The expression is evaluated before every iteration of the loop. If its value is true, the statements inside the loop are executed. If it is false, execution continues with the next statement following the loop. If the value of the Boolean expression is false the first time it is examined, then the loop is skipped in its entirety.

6.5.3. Repeat Loop

The footer of a repeat loop contains a Boolean expression.

```
repeat_loop      ::= REPEAT stmt_list_opt UNTIL expr
```

The expression is evaluated after every iteration of the loop. If its value is false, the statements inside the loop are executed again. If it is true, execution continues with the next statement following the loop. The statements inside a repeat loop are always executed at least once.

6.5.4. Foreach Loop

The header of a foreach loop introduces a new variable called the **index** of the loop.

```
foreach_loop     ::= FOREACH IDENTIFIER IN generator DO
                     stmt_list_opt END
generator        ::= [ expr .. expr ]
                 ::= set
                 ::= selector firstchange
                 ::= REVERSE reversible_gen
reversible_gen   ::= [ expr .. expr ]
                 ::= selector
```

The scope of the index is the statement list inside the loop. The type of the index is determined by the loop's **generator**. A generator can be a range of values, a set expression, a name of a set variable, or a name of a scalar type.

The generator produces a sequence of values for the index. The statements inside the foreach loop are executed once for each value. If the generator is a range of values, then the type of the index will be the base type of the bounds of the range (the bounds must have the same base type). The index takes on the values in the range in ascending or descending order, depending on whether the word REVERSE appears in the loop header. The range may be empty, in which case the loop is skipped in its entirety.

If the generator is a set expression or a variable of a set type, then the type of the index is the base type of the components of the set. The index takes on the values of the set in arbitrary order.

If the generator is the name of a scalar type, then that type is the type of the index. The index takes on the values of the type in ascending or descending order, depending on whether the word REVERSE appears in the loop header.

The value of the index can be examined but not changed by the statements in the loop. It cannot appear on the left-hand side of an assignment, nor can it be passed as a VAR parameter to any procedure or function, nor can it appear among the request parameters of an accept statement or the reply parameters of a connect, call, or receive statement.

6.6. Exit Statement

A exit statement can only appear inside a loop or inner compound statement (not the body of a subroutine, module, or entry). An exit statement causes control to jump to the statement immediately following the loop or compound statement.

```
exit_stmt        ::= EXIT ident_opt
ident_opt        ::= IDENTIFIER
                 ::=
```

Any loop statement or compound statement can be preceded by a label.

```
label_opt        ::= << IDENTIFIER >>
                 ::=
```

The scope of the identifier in a label is the statement list inside the immediately following loop or compound statement. The identifier in an exit statement must have been introduced in a label. Control jumps to the statement immediately

following the labeled statement. If the identifier in the exit statement is missing, then control jumps to the statement immediately following the closest enclosing loop or compound statement.

6.7. Return Statement

A return statement can only appear inside a subroutine.

```
return_stmt     ::= RETURN expr_opt
expr_opt        ::= expr
                ::=
```

If the subroutine is a function, the type of the return expression must be compatible with the type of the function. The function yields the value of the expression, and control returns to the evaluation of the expression in which the function call appeared. If control reaches the end of the body of a function without encountering a return statement, then an error has occurred and execution must halt.

If the subroutine is a procedure, then the return expression must be missing. Control continues with the statement immediately following the procedure call. There is an implicit return statement at the end of the body of every procedure.

6.8. With Statement

A with statement makes it easier and more efficient to access the fields of a record.

```
with_stmt       ::= WITH designator DO stmt_list_opt END
designator      ::= selector firstchange
```

The designator must be the name of a record variable. Within the statement list

of the with statement, the fields of the record can be named directly, without preceding them with the designator and a period. The with statement constitutes a nested scope; any existing meanings for the names of the fields will be hidden.

6.9. Bind and Unbind Statements

The bind statement associates link ends with entries. The unbind statement undoes associations.

```
bind_stmt       ::= BIND expr_list TO ident_list
unbind_stmt     ::= UNBIND expr_list FROM ident_list
```

Each expression in the expression list must either be of type link or else be a set of component base type link. Each identifier in the identifier list must be the name of an entry. Each mentioned link end is bound (unbound) to (from) each mentioned entry. If any of the link values are not valid, then an error has occurred and execution must halt.

Binding and unbinding are idempotent operations when performed by a single thread of control; a thread does no harm by making the same binding twice, or by attempting to break a non-existent binding. Conflicting bindings are a run-time error. If two threads attempt to bind the same link end to different instances of the same entry (same entry lexically, but different environments), or if one or more threads attempt to bind the same link end to different entries with the same name, then an error has occurred and execution must halt.

The purpose of bindings is discussed under execution (section 7) below.

6.10. Await Statement

The await statement is used to suspend execution of the current thread of control until a given condition holds.

await_stmt ::= AWAIT expr

The expression must be of type Boolean. The current thread will not continue until the expression is true. If it is false when first encountered, it must be changed by a different thread.

6.11. Compound Statement

A compound statement is a delimited list of statements with an optional set of exception handlers.

```
compound_stmt   ::= BEGIN stmt_list_opt hand_list_opt END
hand_list_opt   ::= when_clause more_handlers
                ::=
when_clause     ::= WHEN exception more_whens DO stmt_list_opt
more_whens      ::= , exception more_whens
                ::=
more_handlers   ::= when_clause more_handlers
                ::=
exception       ::= expr ident_opt
```

Compound statements comprise the bodies of subroutines, modules, and entries. They may also be nested anywhere a statement can occur.

Each exception handler consists of a series of when clauses and a statement list. As mentioned in section 3.4, an exception is either an expression of type link followed by the name of a built-in exception class, or a name introduced in an exception declaration. The exceptions in the when clauses of a given compound statement need not be distinct. When an exception arises, the first clause that matches the exception will be used. Exceptions are discussed in more detail in section 7.2.

6.12. Raise Statement

Some exceptions occur spontaneously in the course of communication on links. Others are caused by execution of the raise statement.

raise_stmt ::= RAISE exception

An exception associated with a link end is raised in the current thread of control. An exception introduced by an exception declaration is raised in each thread with an active handler for it.

6.13. Input/Output Statements

Input and output statements read and write Ascii data on the standard input and output streams. In the Charlotte implementation, these streams connect to the (possibly virtual) console terminal of the local node.

```
io              ::= WRITE ( expr_list )
                ::= READ ( expr des_list_opt )
des_list_opt    ::= designator_list
                ::=
designator_list ::= designator des_list_tail
des_list_tail   ::= , designator des_list_tail
                ::=
```

The parameters of read and write have the same format as those of the *scanf* and *printf* routines in C. The first argument must be a string constant or an array whose elements have base type char. The rest of the arguments must be scalars or strings. The second and subsequent arguments to read are automatically passed by reference.

6.14. Communication Statements

Communication statements use links to exchange messages with remote processes.

```
communication   ::= connect_stmt
                ::= call_stmt
                ::= accept_stmt
                ::= send_stmt
                ::= receive_stmt
```

6.14.1. Connect and Call Statements

The connect statement requests a remote operation. The call statement invokes a local operation.

```
connect_stmt    ::= CONNECT IDENTIFIER call_args_opt ON expr
call_stmt       ::= CALL IDENTIFIER call_args_opt
call_args_opt   ::= ( call_args )
                ::=
call_args       ::= expr_list | des_list_opt
                ::= | designator_list
```

The identifier following the word CONNECT or CALL must be the name of an entry. The final expression of a connect statement must have type link.

The thread of control that executes a connect or call statement is called a **client**. The client creates a **request** message from the actual parameters of the expression list, sends the message, and waits for a **reply** message. The reply will contain new values for the actual parameters in the designator list. The request actual parameters must agree in number and order with the formal parameters of the entry whose name follows the word CONNECT or CALL. Their types must be compatible with those of the formals. The reply actual parameters must be the names of variables. They must agree in number and order, and be compatible,

with the reply types of the entry.

6.14.2. Accept Statement

The accept statement allows a thread of control to serve a request from some other process for a remote operation.

```
accept_stmt     ::= ACCEPT IDENTIFIER d_arg_opt ON expr ;
                    o_s_list_opt reply
o_s_list_opt    ::= other_stmt ; o_s_list_opt
                ::=
d_arg_opt       ::= ( designator_list )
                ::=
reply           ::= REPLY e_arg_opt
```

The identifier following the word ACCEPT must be the name of an entry. The expression following the word ON must have type link.

The thread of control that executes an accept statement is called a **server**. The server waits for a request message from a client on the other end of the referenced link. When such a message arrives, it will contain new values for the actual parameters in the designator list. The parameters in that list must agree in number and order, and be compatible, with the parameters of the entry whose name follows the word ACCEPT.

The server executes the statement list and returns a reply message to the client. The actual parameters following the word REPLY must agree in number and order, and be compatible, with the reply types of the entry whose name follows the word ACCEPT. The actuals are packaged together to form the reply message. They are returned to the client on the link specified after the word ON — the same link on which the request message arrived.

The syntax of the portion of an accept statement beginning with the word REPLY is a valid statement in and of itself; therefore the statements inside the *accept* cannot include a *reply*.

6.14.3. Reply Statement

Accept statements provide for the **explicit** receipt of requests for remote operations. Entries provide for **implicit** receipt. Within the body of an entry, the reply portion of an accept statement can appear by itself. It~ actual parameters must agree in number and order, and be ~ompatible, with the return types of the entry in which the reply statement occurs. The reply message is returned to the client on the same link on which the request message arrived.

If control reaches the end of an entry without replying, or if the thread of control executing the entry attempts to reply more than once, then an error has occurred and execution must halt.

6.14.4. Send Statement

The send statement allows a thread to escape the normal checking of operation names and message types.

```
send_stmt       ::= SEND designator length_opt enclosure_opt ON expr
length_opt      ::= < term >
                ::=
enclosure_opt   ::= WITH expr
                ::=
```

The designator following the word SEND must be the name of a variable. The expression following the word ON must have type link. A sequence of bytes, beginning at the location of the variable, are sent on the referenced link. The length option indicates the number of bytes to be sent. If missing, the size of the

variable is assumed (and the bytes that are sent are precisely the contents of the variable). No interpretation is implied for the transferred bytes; in particular, link variables that happen to lie among them do *not* cause the link ends they reference to be moved. In the Charlotte implementation, a single enclosure can be attached to the message by means of an optional *with* clause.

The thread of control that executes a send statement blocks until the message is received by some thread in the process on the other end of the link. It does *not* wait for a reply message.

6.14.5. Receive Statement

The receive statement is the counterpart of the send statement. It allows a thread to escape the normal checking on messages.

```
receive_stmt    ::= RECEIVE designator length_opt enclosure_opt
                    ON expr
```

The designator following the word RECEIVE must be the name of a variable. The expression following the word ON must have type link. A sequence of bytes is received on the referenced link and stored in memory beginning at the location of the variable. The length option indicates the number of bytes to be received. If missing, the size of the variable is assumed (and the bytes that are received constitute new contents for the variable). The enclosure option is provided for the benefit of the Charlotte implementation. It must be the name of a variable of type link. The contents of the link variable are changed to reference the link end that was enclosed in the message. If no end was enclosed, the contents of the link variable are changed to nolink.

The thread of control that executes a receive statement blocks until a message arrives.

6.14.6. Communication Rules

Messages sent in the same direction on the same link are guaranteed to arrive in order. Messages sent on different links are not, even if they involve the same pair of processes. Similarly, the cleanup of the far end of a link that is destroyed locally may occur in arbitrary order with respect to the destruction or arrival of messages on other links.

If any of the following rules is broken, then an error has occurred and execution must halt.

(1) For all communication statements, the value of the expression that follows the word ON must reference a valid link.

(2) A link end that is bound to an entry, or that is being used in a connect, accept, or reply statement may not simultaneously be enclosed in a message.

(3) A link end that is bound to an entry, or that is being used in a connect, accept, or reply statement may not simultaneously be used in a send or receive statement.

Rule 3 is not enforced correctly by the Charlotte implementation. There are (unlikely) circumstances under which invalid communication will be allowed or valid communication forbidden. In general, a program is safe if it avoids using *send* and *receive* on links that may occasionally have threads executing connect statements on both ends simultaneously.

6.14.7. Enclosures

There are no limitations on the data types that can appear in the argument lists of connect, accept, and reply statements. In particular, references to links

and data structures that contain references to links can be transferred from one process to another.

If a link variable that references a valid link is enclosed in a request or reply message, then the end of the link that it references is *moved* to the receiving process. The contents of the link variable are changed in the receiving process to be a valid reference to the moved end of the link. A single link variable can also be enclosed in a send statement, but only by means of the with clause (section 6.14.4).

A link end that is enclosed in a message becomes inaccessible in the sending process, even if communication is interrupted by an exception. Link variables that referenced the end are now dangling references; their contents are no longer valid.

A process can own both ends of a link. If it sends a message to itself on that link, references to any enclosures still become invalid, just as if they had been sent to another process. Link variables that refer to enclosures in call statements or in replies from called entries do *not* become dangling; they remain valid.

7. Execution

A LYNX program is a collection of modules. Modules nest. The syntax for outermost modules differs slightly from that of other modules. Each outermost module is inhabited by a single process.

```
process_list    ::= process . process_list
                ::=
process         ::= MODULE IDENTIFIER in_args_opt ;
                    dec_pt cpd_stmt_opt IDENTIFIER
```

An outermost module has no import and export lists. Its arguments must have built-in types. Links in the argument list provide the means for a process to communicate with the rest of the world.

A process begins execution with a single thread of control. The task of that thread is to execute the initialization code of the process's outermost module. Before doing so, the thread recursively executes the initialization code of any nested modules. In general, a thread of control executes the initialization code of a module immediately before executing the body of the subroutine, module, or entry in which that module is declared.

New threads of control are created by instantiating entries. Entries are instantiated by call statements and by the arrival of messages on link ends bound to entries.

The threads in a process do not execute in parallel. A process continues with a given thread until it **blocks**. (Blocking statements are listed in section 7.1.) It then switches context to another thread. If no other thread is runnable, the process waits for an **event**. An event is the completion of an outstanding connect or reply statement, or the arrival of a request on a link end that is bound to an entry or for which there are outstanding accept statements. If no events are expected, then **deadlock** has occurred and execution must halt. *Events only complete when all threads are blocked.*

The completion of an event always allows some thread to continue execution. Only one event completes at a time. The nature of the event determines which thread runs next. If a connect or reply statement has completed, the thread that executed that statement can continue. If a request arrives on a link end for which there are outstanding accept statements or bindings to entries, then

the contents of the request are examined.

If the requested operation matches the name of an entry in one of the accept statements, then the thread that executed that accept statement can continue. If the requested operation matches the name of an entry in one of the bindings, then a new thread of control is created. That thread begins execution in the appropriate entry with initial values for its parameters taken from the message. If outstanding accept statements or bindings exist, but the requested operation matches none of them, then a built-in exception of class INVALID_OP is raised at the connect statement on the other end of the link and the local process waits for another event.

As mentioned in section 4, the meanings of identifiers visible to a given thread of control come and go in LIFO order. Likewise, the management of storage for the variables accessible to the thread can be performed in LIFO order. Variables declared in an outermost module are created when their process is created. Parameters and variables declared local to a subroutine or entry are created when control enters the body of their block. Variables declared immediately inside a non-outermost module are created when control enters the body of the closest enclosing subroutine, entry, or outermost module. Different instantiations of the same subroutine or entry do *not* share local variables.

Since a process may have many suspended threads of control at a given point in time, the variables of a process as a whole cannot be managed on a stack. The creation of a new thread of control in an entry creates a new branch in a run-time environment *tree*. The environment of a thread created with a call statement is similar to that of a procedure; in addition to (new) local variables, it shares the variables in enclosing blocks with its caller. The environment of a

thread created in response to a message on a bound link is the same as it would have been if the entry in question had been called locally at the point the bind statement was executed.

Control is not allowed to return from a subroutine whose local variables are still accessible to other threads of control or to potential threads that might be created in response to incoming messages. Similarly, a thread does not terminate when it reaches the end of the body of its entry; it too waits for nested threads to finish. A process terminates only after all its threads have finished. A thread that is waiting for nested threads does so at the very bottom of the block, *after* the word END. Exception handlers for the block are no longer active.

7.1. Blocking Statements

The absence of asynchronous context switches allows the programmer to assume that data structures remain consistent until the current thread of control blocks. A context switch between the threads of a process can occur

(1) at every connect, call, accept, and reply statement,

(2) at every await statement,

(3) whenever the current thread terminates, and

(4) whenever control reaches the end of a subroutine, entry, or outermost module whose local variables remain accessible to other threads or potential threads.

In the absence of exceptions, a thread that resumes execution after a context switch continues with the statement immediately following the statement that blocked. Functions must not contain blocking statements or calls to subroutines whose execution may lead to a blocking statement.

7.2. Exception Handling

Exceptions interrupt the normal flow of control. They come in two varieties.

Built-in exceptions are associated with links. In the process of communication on link end **L**, the following exceptions may arise:

L INVALID_OP

A remote operation was requested, but it was not among those for which there were *accepts* or bindings in the process on the far end of **L**.

L TYPE_CLASH

A remote operation was requested, and the process on the far end of **L** was willing to serve it, but the two processes disagreed on the number, order, or types of the request or reply parameters.

L LOCAL_DESTROYED

The link end referenced by **L** was destroyed by a thread of control in the local process.

L REMOTE_DESTROYED

The other end of the link referenced by **L** was destroyed by a thread in the process that owned it.

L EXC_REPLY

A remote operation had started, but the thread of control that was serving it felt an exception that prevented it from replying.[16]

[16] There is no corresponding exception for a server whose client feels a locally-defined exception before it can receive its reply. When a reply statement completes without exception, a server can assume that the reply message was successfully delivered if and only if the client thread was still alive within the process on the far end of the link. The server *can* be sure that the client's *process*

L LENGTH_CLASH

An unchecked *send* or *receive* was attempted, but the receiver wanted fewer bytes than the sender sent. The built-in function ACTUAL_LENGTH will return the number of bytes successfully transferred. This number will be the smaller of the lengths expected by the two processes.

Exceptions occur only when all threads are blocked. Built-in exceptions are raised in the thread in which they arise. The handlers of the closest enclosing compound statement are examined in order to see if one of them matches the exception that arose. If one does, then the thread is moved to the beginning of the matching handler and is ready to continue. The handler will be executed *in place of* the portion of the compound statement that had yet to be executed when the exception occurred.

If the closest enclosing compound statement has no handlers, or if none of them matches the exception, then the exception **propagates** to the handlers of the nest enclosing compound statement. If the propagation reaches the compound statement comprising the body of a subroutine, then the exception is raised at the subroutine's point of call, and propagation continues. Any nested threads that still have access to the local variables of the subroutine are aborted (recursively). Likewise any bindings that might create such threads are broken.

The propagation of an exception stops when an appropriate handler is found or when the body of an entry or outermost module is reached. A thread with no appropriate handler is aborted. If propagation escapes the scope of an accept statement, or if an exception remains unhandled in the body of an entry that has not yet replied, then a built-in exception of class EXC_REPLY is raised at the

---

was alive and that the link between them was still intact.

corresponding connect statement in the process on the other end of the link.

In the absence of exceptions, when all threads are blocked, the occurrence of an event allows exactly one thread to continue. With exceptions, however, more than one thread may be unblocked at once. When a link is destroyed, for example, all threads waiting for the completion of communication on the same end of that link are moved to the beginning of their handlers simultaneously, and an arbitrary one is chosen to continue first.

Both built-in and programmer-defined exceptions can arise from use of the raise statement. A built-in exception is raised as if it had occurred in communication in the current thread of control. By contrast, a programmer-defined exception is raised in *all and only* those threads that have an active handler for it. Once raised in a thread, a programmer-defined exception propagates like a built-in exception. The only difference is that the propagation will always encounter an appropriate handler by the time it reaches the compound statement in which the thread originated.

When a connect, accept, or reply statement is interrupted by a programmer-defined exception, the language makes no guarantee about whether or not the requested communication will have occurred. Any of the following conditions may hold.

connect

1) The operation may not have started. The process at the other end of the link does not know anything has happened. 2) The request may have been received by the process at the far end of the link. It is now being served. The reply message will be discarded when it arrives. 3) The operation may have completed. The reply message will have been discarded.

accept

1) The operation may not have started. The process at the other end of the link does not know anything has happened. 2) A request may have been received. The connected thread (if it still exists) in the process at the other end of the link will feel a built-in exception of class EXC_REPLY.

reply

1) The operation may not have completed. The connected thread (if it still exists) in the process at the other end of the link will feel a built-in exception of class EXC_REPLY. If the server thread attempts to reply again, then an error has occurred and execution must stop. 2) The operation may have completed. The process at the other end of the link does not know anything has happened.

In the case of connect and reply, link ends that were enclosed (or were to have been enclosed) are no longer accessible to the sending process.

7.3. Message Type Checking

Since the client and server involved in a remote operation will in general be in different processes, they will share no declarations. Run-time checking is necessary to assure that they agree on the number, order, and **structural equivalence** of request and reply parameters.

Structural equivalence is a weaker check than the notion of compatible types used in the rest of the language. The built-in types are of course equivalent in every process. Enumeration types are equivalent if they have the same number of values. Subrange types are equivalent if they have the same bounds and the same (built-in) base type. Array types are equivalent if they have

equivalent index and element types. Record types are equivalent if their fields have equivalent types and occur in the same order.

If a client requests an operation that the process on the other end of the link is willing to serve, but the server would disagree about the number, order, or structure of the parameters of the request *or* reply messages, then a built-in exception of class TYPE_CLASH is raised in the client. The server continues to wait for a valid, matching request.

8. Pre-defined Identifiers

The following identifiers are pre-defined.

types:	Boolean, integer, char, link
constants:	true, false, nolink
exception classes:	TYPE_CLASH, INVALID_OP, LENGTH_CLASH, EXC_REPLY, LOCAL_DESTROYED, REMOTE_DESTROYED
functions:	newlink, valid, curlink, ACTUAL_LENGTH
procedures:	destroy

The types, constants, and exception classes have been discussed elsewhere.

The function "newlink" takes a single reference parameter of type link and yields a value of type link. The parameter and function value return references to the two ends of a new link, created as a side effect.

The function "valid" takes a single value parameter of type link and yields a value of type Boolean. The value indicates whether the parameter accesses an end of a currently valid link that can be used in communication or bindings.

The function "curlink" takes no parameters. It returns a value of type link. The value is a reference to the link on which the request message arrived for the closest lexically-enclosing entry (*not* the original entry for the current

thread of control). If there is no enclosing entry, or if the closest enclosing entry was invoked locally with a call statement, then curlink yields nolink.

The function "ACTUAL_LENGTH" takes no parameters. It returns a value of type integer. If ACTUAL_LENGTH is called before the first context switch after the completion of a send or receive statement, then the value is the number of bytes actually transferred. In other circumstances, ACTUAL_LENGTH returns garbage. ACTUAL_LENGTH is intended for use in code immediately following a receive statement or in handlers for LENGTH_CLASH exceptions.

The procedure "destroy" takes a single value parameter of type link. It destroys the corresponding link. Variables referencing either end of the link (in any process) become invalid. An attempt to destroy a nil or dangling link is a no-op.

9. Collected Syntax

The following is an LL(1) grammar for LYNX. *Process_list* is the start symbol. The notation

```
A           ::= B | C
```

is shorthand for

```
A           ::= B
            ::= C
```

Epsilon (ϵ) denotes the empty string.

```
accept_stmt     ::= ACCEPT IDENTIFIER d_arg_opt ON expr ;
                    o_s_list_opt reply
arg_list_opt    ::= ( mode formal more_m_formals ) | ϵ
```



```
array_type      ::= ARRAY type OF type
await_stmt      ::= AWAIT expr
bind_stmt       ::= BIND expr_list TO ident_list
                ::= UNBIND expr_list FROM ident_list
body            ::= dec_pt compound_stmt IDENTIFIER
                ::= FORWARD | EXTERNAL | REMOTE
call_args       ::= expr_list | des_list_opt | | designator_list
call_args_opt   ::= ( call_args ) | ϵ
call_stmt       ::= CALL IDENTIFIER call_args_opt
case_list_opt   ::= { component_list } stmt_list_opt case_list_opt | ϵ
case_stmt       ::= CASE expr OF case_list_opt default_opt END
changeover      ::= [ expr ] designator_tail
                ::= : IDENTIFIER designator_tail
communication   ::= connect_stmt | call_stmt | accept_stmt
                ::= send_stmt | receive_stmt
comp_list_opt   ::= component_list | ϵ
comp_list_tail  ::= , component comp_list_tail | ϵ
component       ::= expr component_tail
component_list  ::= component comp_list_tail
component_tail  ::= .. expr | ϵ
compound_stmt   ::= BEGIN stmt_list_opt hand_list_opt END
connect_stmt    ::= CONNECT IDENTIFIER call_args_opt ON expr
const_dec       ::= IDENTIFIER = expr ; | ;
const_dec_tail  ::= const_dec const_dec_tail | ϵ
constant        ::= CHARCONST | STRINGCONST | NUMBER
cpd_stmt_opt    ::= compound_stmt | END
d_arg_opt       ::= ( designator_list ) | ϵ
dec_pt          ::= declaration dec_pt | ϵ
declaration     ::= CONST const_dec const_dec_tail
                ::= TYPE type_dec type_dec_tail
                ::= VAR variable_dec var_dec_tail
                ::= EXCEPTION ident_list ;
                ::= subroutine ; | entry ; | module ;
default_opt     ::= OTHERWISE stmt_list_opt | ϵ
des_list_opt    ::= designator_list | ϵ
des_list_tail   ::= , designator des_list_tail | ϵ
designator      ::= selector firstchange
designator_list ::= designator des_list_tail
designator_tail ::= . IDENTIFIER designator_tail
                ::= changeover | ϵ
e_arg_opt       ::= ( expr_list ) | ϵ
```

```
else_opt        ::= ELSE stmt_list_opt | ϵ
elsif_list_opt  ::= ELSIF expr THEN stmt_list_opt elsif_list_opt | ϵ
enclosure_opt   ::= WITH expr | ϵ
entry           ::= ENTRY IDENTIFIER in_args_opt out_types_opt ; body
enum_type       ::= ( ident_list )
exception       ::= expr ident_opt
exit_stmt       ::= EXIT ident_opt
export_pt       ::= EXPORT ident_list ; | ϵ
expr            ::= term expr_tail
expr_list       ::= expr expr_list_tail
expr_list_tail  ::= , expr_list | ϵ
expr_opt        ::= expr | ϵ
expr_tail       ::= rel_op term expr_tail | ϵ
factor          ::= NOT factor | - factor | constant
                ::= set | ( expr ) | selector sel_fac_tail
field           ::= ident_list : type ; | ;
                ::= CASE IDENTIFIER : type OF vnt_list_opt END ;
field_list_opt  ::= field field_list_opt | ϵ
firstchange     ::= changeover | ϵ
foreach_loop    ::= FOREACH IDENTIFIER IN generator DO
                    stmt_list_opt END
forever_loop    ::= LOOP stmt_list_opt END
formal          ::= ident_list : IDENTIFIER
formal_tail     ::= ; formal formal_tail | ϵ
fun_type_opt    ::= : IDENTIFIER | ϵ
generator       ::= [ expr .. expr ] | selector firstchange
                ::= set | REVERSE reversible_gen
hand_list_opt   ::= when_clause more_handlers | ϵ
id_list_tail    ::= , ident_list | ϵ
ident_list      ::= IDENTIFIER id_list_tail
ident_opt       ::= IDENTIFIER | ϵ
if_stmt         ::= IF expr THEN stmt_list_opt elsif_list_opt else_opt END
import_pt       ::= IMPORT ident_list ; | ϵ
in_args_opt     ::= ( formal formal_tail ) | ϵ
io              ::= WRITE ( expr_list )
                ::= READ ( expr des_list_opt )
label_opt       ::= << IDENTIFIER >> | ϵ
labeled_stmt    ::= loop_stmt | compound_stmt
length_opt      ::= < term > | ϵ
loop_stmt       ::= while_loop | foreach_loop
                ::= repeat_loop | forever_loop
```



```
mode            ::= VAR | CONST | ϵ
module          ::= MODULE IDENTIFIER ; import_pt export_pt
                    dec_pt cpd_stmt_opt IDENTIFIER
more_handlers   ::= when_clause more_handlers | ϵ
more_m_formals  ::= ; mode formal more_m_formals | ϵ
more_whens      ::= , exception more_whens | ϵ
o_s_list_opt    ::= other_stmt ; o_s_list_opt | ϵ
other_op        ::= OR | AND | + | - | * | / | MOD
other_stmt      ::= label_opt labeled_stmt | communication | io
                ::= bind_stmt | if_stmt | case_stmt | exit_stmt
                ::= with_stmt | return_stmt | await_stmt | raise_stmt
                ::= selector sel_stmt_tail | ϵ
out_types_opt   ::= : ident_list | ϵ
process         ::= MODULE IDENTIFIER in_args_opt ;
                    dec_pt cpd_stmt_opt IDENTIFIER
process_list    ::= process . process_list | ϵ
raise_stmt      ::= RAISE exception
receive_stmt    ::= RECEIVE designator length_opt enclosure_opt ON expr
record_type     ::= RECORD field_list_opt END
rel_op          ::= IN | ~ | ~>
                ::= = | <> | < | <= | > | >=
repeat_loop     ::= REPEAT stmt_list_opt UNTIL expr
reply           ::= REPLY e_arg_opt
return_stmt     ::= RETURN expr_opt
reversible_gen  ::= [ expr .. expr ] | selector
sel_fac_tail    ::= changeover | ( expr_list ) | ϵ
sel_stmt_tail   ::= firstchange := expr | e_arg_opt
selector        ::= IDENTIFIER selector_tail
selector_tail   ::= . IDENTIFIER selector_tail | ϵ
send_stmt       ::= SEND designator length_opt enclosure_opt ON expr
set             ::= { comp_list_opt }
set_type        ::= SET OF type
stmt            ::= reply | other_stmt
stmt_list_opt   ::= stmt ; stmt_list_opt | ϵ
subr_type       ::= [ expr .. expr ]
subroutine      ::= PROCEDURE IDENTIFIER arg_list_opt ; body
                ::= FUNCTION IDENTIFIER arg_list_opt fun_type_opt
                    ; body
term            ::= factor term_tail
term_tail       ::= other_op factor term_tail | ϵ
type            ::= IDENTIFIER | enum_type | subr_type
```

```
                ::= array_type | record_type | set_type
type_dec        ::= IDENTIFIER = type ; | ;
type_dec_tail   ::= type_dec type_dec_tail | ϵ
var_dec_tail    ::= variable_dec var_dec_tail | ϵ
variable_dec    ::= ident_list : type ; | ;
variant         ::= { component_list } field_list_opt
vnt_list_opt    ::= variant vnt_list_opt | ϵ
when_clause     ::= WHEN exception more_whens DO stmt_list_opt
while_loop      ::= WHILE expr DO stmt_list_opt END
with_stmt       ::= WITH designator DO stmt_list_opt END
```


REFERENCES

[1] Allchin, J. E. and M. S. McKendry, "Synchronization and Recovery of Actions," *ACM Operating Systems Review* 19:1 (January 1985), pp. 32-45. Originally presented at the Second ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, 17-19 August, 1983.

[2] Almes, G. T., A. P. Black, E. D. Lazowska, and J. D. Noe, "The Eden System: A Technical Review," Technical Report 83-10-05, Department of Computer Science, University of Washington, October 1983.

[3] Andrews, G. R. and J. R. McGraw, "Language Features for Process Interaction," *Proceedings of an ACM Conference on Language Design for Reliable Software*, 28-30 March 1977, pp. 114-127. In *ACM SIGPLAN Notices* 12:3 (March 1977).

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

[5] Andrews, G. R., "The Distributed Programming Language SR — Mechanisms, Design and Implementation," *Software — Practice and Experience* 12 (1982), pp. 719-753.

[6] Andrews, G. R. and F. B. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys* 15:1 (March 1983), pp. 3-44.

[7] Artsy, Y., H.-Y. Chang, and R. Finkel, "Charlotte: Design and Implementation of a Distributed Kernel," Computer Sciences Technical Report #554, University of Wisconsin - Madison, August 1984.

[8] Baiardi, F., L. Ricci, and M. Vanneschi, "Static Checking of Interprocess Communication in ECSP," *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, 17-22 June 1984, pp. 290-299. In *ACM SIGPLAN Notices* 19:6 (June 1984).

[9] Ball, J. E., G. J. Williams, and J. R. Low, "Preliminary ZENO Language Description," *ACM SIGPLAN Notices* 14:9 (September 1979), pp. 17-34.

[10] Baskett, F., J. H. Howard, and J. T. Montague, "Task Communication in Demos," *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, November 1977, pp. 23-31.

[11] Beech, D., "A Structural View of PL/I," *Computing Surveys* 2:1 (March 1970), pp. 33-64.

[12] Bernstein, A. J., "Output Guards and Nondeterminism in 'Communicating Sequential Processes'," *ACM TOPLAS* 2:2 (April 1980), pp. 234-238.

[13] Bernstein, A. J. and J. R. Ensor, "A Modula Based Language Supporting Hierarchical Development and Verification," *Software — Practice and Experience* 11 (1981), pp. 237-255.

[14] Birtwistle, G. M., O.-J. Dahl, B. Myhrhaug, and K. Nygaard, *SIMULA Begin*, Auerback Press, Philadelphia, 1973.

[15] Black, A. P., "An Asymmetric Stream Communication System," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, 10-13 October 1983, pp. 4-10. In *ACM Operating Systems Review* 17:5.

[16] Bobrow, D. G. and B. Raphael, "New Programming Languages for Artificial Intelligence Research," *Computing Surveys* 6:3 (September 1974), pp. 153-174.

[17] Bos, J. van den, "Input Tools — A New Language for Input-Driven Programs," *Proceedings of the European Conference on Applied Information Technology*, IFIP, 25-28 September 1979, pp. 273-279. Published as *EURO IFIP 79*, North-Holland, Amsterdam, 1979.

[18] Bos, J. van den, R. Plasmeijer, and J. Stroet, "Process Communication Based on Input Specifications," *ACM TOPLAS* 3:3 (July 1981), pp. 224-250.

[19] Brinch Hansen, P., "Structured Multi-programming," *CACM* 15:7 (July 1972), pp. 574-578.

[20] Brinch Hansen, P., *Operating System Principles*, Prentice-Hall, 1973.

[21] Brinch Hansen, P., "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering* SE-1:2 (June 1975), pp. 199-207.

[22] Brinch Hansen, P., "Distributed Processes: A Concurrent Programming Concept," *CACM* 21:11 (November 1978), pp. 934-941.

[23] Brinch Hansen, P., "The Design of Edison," Technical Report, University of Southern California Computer Science Department, September 1980.

[24] Brinch Hansen, P., "Edison: A Multiprocessor Language," Technical Report, University of Southern California Computer Science Department, September 1980.

[25] Browne, J. C., J. E. Dutton, V. Fernandes, A. Palmer, J. Silverman, A. R. Tripathi, and P.-S. Wang, "Zeus: An Object-Oriented Distributed Operating System for Reliable Applications," *Proceedings of the 1984 ACM Annual Conference*, 8-10 October 1984, pp. 179-188.

[26] Buckley, G. N. and A. Silberschatz, "An Effective Implementation for the Generalized Input-Output Construct of CSP," *ACM TOPLAS* 5:2 (April 1983), pp. 223-235.

[27] Burger, W. F., N. Halim, J. A. Pershing, F. N. Parr, R. E. Strom, and S. Yemini, "Draft NIL Reference Manual," RC 9732 (#42993), I.B.M. T. J. Watson Research Center, December 1982.

[28] Campbell, R. H. and A. N. Habermann, "The Specification of Process Synchronization by Path Expressions," pp. 89-102 in *Operating Systems*, Lecture Notes in Computer Science #16, ed. C. Kaiser, Springer-Verlag, Berlin, 1974.

[29] Cashin, P., "Inter-Process Communication," Technical Report 8005014, Bell-Northern Research, 3 June 1980.

[30] Cheriton, D. R. and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, 10-13 October 1983, pp. 128-139. In *ACM Operating Systems Review* 17:5.

[31] Cook, R. P., "*Mod--A Language for Distributed Programming," *IEEE Transactions on Software Engineering* SE-6:6 (November 1980), pp. 563-571.

[32] Cook, R. P., "The StarMod Distributed Programming System," *IEEE COMPCON Fall 1980*, September 1980, pp. 729-735.

[33] Courtois, P. J., F. Heymans, and D. L. Parnas, "Concurrent Control with 'Readers' and 'Writers'," *CACM* 14:10 (October 1971), pp. 667-668.

[34] DeWitt, D. J., R. Finkel, and M. Solomon, "The CRYSTAL Multicomputer: Design and Implementation Experience," Computer Sciences Technical Report #553, University of Wisconsin - Madison, September 1984.

[35] Dijkstra, E. W., "Co-operating sequential processes," pp. 43-112 in *Programming Languages*, ed. F. Genuys, Academic Press, London, 1968.

[36] Dijkstra, E. W., "Hierarchical Ordering of Sequential Processes," pp. 72-93 in *Operating Systems Techniques*, A. P. I. C. Studies in Data Processing #9, ed. C. A. R. Hoare and R. H. Perrott, Academic Press, London, 1972. Also *Acta Informatica* 1 (1971), pp 115-138.

[37] Dijkstra, E. W., "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," *CACM* 18:8 (August 1975), pp. 453-457.

[38] Ellis, C. S., J. A. Feldman, and J. E. Heliotis, "Language Constructs and Support Systems for Distributed Computing," *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 18-20 August 1982, pp. 1-9.

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

[40] Finkel, R., R. Cook, D. DeWitt, N. Hall, and L. Landweber, "Wisconsin Modula: Part III of the First Report on the Crystal Project," Computer Sciences Technical Report #501, University of Wisconsin - Madison, April 1983.

[41] Finkel, R., M. Solomon, D. DeWitt, and L. Landweber, "The Charlotte Distributed Operating System: Part IV of the First Report on the Crystal Project," Computer Sciences Technical Report #502, University of Wisconsin - Madison, October 1983.

[42] Finkel, R. and U. Manber, "DIB: A Distributed Implementation of Backtracking," submitted to the *Fifth International Conference on Distributed Computing Systems*, September 1984.

[43] Finkel, R. A., "Tools for Parallel Programming," Appendix B of Second Report, Wisconsin Parallel Array Computer (WISPAC) Research Project, University of Wisconsin Electrical and Computer Engineering Report #80-27, August 1980.

[44] Finkel, R. A. and M. H. Solomon, "The Arachne Distributed Operating System," Computer Sciences Technical Report #439, University of Wisconsin - Madison, 1981.

[45] Fischer, C. N., D. R. Milton, and S. B. Quiring, "Efficient LL(1) Error Correction and Recovery Using Only Insertions," *Acta Informatica* 13:2 (1980), pp. 141-154.

[46] Fishburn, J. P., "An Analysis of Speedup in Parallel Algorithms," Ph. D. thesis, Computer Sciences Technical Report #431, University of Wisconsin - Madison, May 1981.

[47] Gelernter, D. and A. Bernstein, "Distributed Communication via Global Buffer," *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 18-20 August 1982, pp. 10-18.

[48] Gelernter, D., "Dynamic Global Name Spaces on Network Computers," *Proceedings of the 1984 International Conference on Parallel Processing*, 21-24 August, 1984, pp. 25-31.

[49] Gelernter, D., "Generative Communication in Linda," *ACM TOPLAS* 7:1 (January 1985), pp. 80-112.

[50] Ghezzi, C. and M. Jazayeri, *Programming Language Concepts*, John Wiley and Sons, New York, 1982.

[51] Good, D. I., R. M. Cohen, and J. Keeton-Williams, "Principles of Proving Concurrent Programs in Gypsy," *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, 29-31 January 1979, pp. 42-52.

[52] Habermann, A. N., "Synchronization of Communicating Processes," *CACM* 15:3 (March 1972), pp. 171-176.

[53] Habermann, A. N., "On the Timing Restrictions of Concurrent Processes," *Fourth Annual Texas Conference on Computing Systems*, 17-18 November 1975, pp. 1A.3.1-1A.3.6.

[54] Haddon, B. K., "Nested Monitor Calls," *ACM Operating Systems Review* 11:4 (October 1977), pp. 18-23.

[55] Herlihy, M. and B. Liskov, "Communicating Abstract Values in Messages," Computation Structures Group Memo 200, Laboratory for Computer Science, MIT, October 1980.

[56] Hoare, C. A. R., "Towards a Theory of Parallel Programming," pp. 61-71 in *Operating Systems Techniques*, A. P. I. C. Studies in Data Processing #9, ed. C. A. R. Hoare and R. H. Perrott, Academic Press, London, 1972.

[57] Hoare, C. A. R., "Monitors: An Operating Systems Structuring Concept," *CACM* 17:10 (October 1974), pp. 549-557.

[58] Hoare, C. A. R., "Communicating Sequential Processes," *CACM* 21:8 (August 1978), pp. 666-677.

[59] Holt, R. C., G. S. Graham, E. D. Lazowska, and M. A. Scott, "Announcing CONCURRENT SP/k," *ACM Operating Systems Review* 12:2 (April 1978), pp. 4-7.

[60] Holt, R. C., G. S. Graham, E. D. Lazowska, and M. A. Scott, *Structured Concurrent Programming with Operating Systems Applications*, Addison-Wesley, 1978.

[61] Holt, R. C., "A Short Introduction to Concurrent Euclid," *ACM SIGPLAN Notices* 17:5 (May 1982), pp. 60-79.

[62] Howard, J. H., "Signaling in Monitors," *Proceedings of the Second International Conference on Software Engineering*, 13-15 October 1976, pp. 47-52.

[63] Ichbiah, J. D., J. G. P. Barnes, J. C. Heliard, B. Krieg-Brueckner, O. Roubine, and B. A. Wichmann, "Rationale for the Design of the ADA Programming Language," *ACM SIGPLAN Notices* 14:6 (June 1979).

[64] Jazayeri, M., C. Ghezzi, D. Hoffman, D. Middleton, and M. Smotherman, "CSP/80: A Language for Communicating Sequential Processes," *IEEE COMPCON Fall 1980*, 23-25 September 1980, pp. 736-740.

[65] Jensen, K. and N. Wirth, *Pascal User Manual and Report*, Lecture Notes in Computer Science #18, Springer-Verlag, Berlin, 1974.

[66] Kahn, G. and D. B. MacQueen, "Coroutines and Networks of Parallel Processes," pp. 993-998 in *Information Processing 77*, ed. B. Gilchrist, North-Holland, 1977. Proceedings of the 1977 IFIP Congress, Toronto, 8-12 August 1977.

[67] Kaubisch, W. H., R. H. Perrott, and C. A. R. Hoare, "Quasiparallel Programming," *Software — Practice and Experience* 6 (1976), pp. 341-356.

[68] Keedy, J. L., "On Structuring Operating Systems with Monitors," *ACM Operating Systems Review* 13:1 (January 1979), pp. 5-9.

[69] Kepecs, J., "SODA: A Simplified Operating System for Distributed Applications," Ph. D. Thesis, University of Wisconsin - Madison, January 1984. Published as Computer Sciences Technical Report #527, by J. Kepecs and M. Solomon.

[70] Kernighan, B. W. and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, 1978.

[71] Kessels, J. L. W., "An Alternative to Event Queues for Synchronization in Monitors," *CACM* 20:7 (July 1977), pp. 500-503.

[72] Kral, J., "The Equivalence of Modes and the Equivalence of Finite Automata," *ALGOL Bulletin* 35 (March 1973), pp. 34-35.

[73] Lampson, B. W., J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek, "Report On The Programming Language Euclid," *ACM SIGPLAN Notices* 12:2 (February 1977).

[74] Lampson, B. W. and D. D. Redell, "Experience with Processes and Monitors in Mesa," *CACM* 23:2 (February 1980), pp. 105-117.

[75] Lauer, H. C. and R. M. Needham, "On the Duality of Operating System Structures," *ACM Operating Systems Review* 13:2 (April 1979), pp. 3-19. Originally presented at the Second International Symposium on Operating Systems, October 1978.

[76] LeBlanc, R. J. and C. N. Fischer, "On Implementing Separate Compilation in Block-Structured Languages," *Proceedings of the SIGPLAN Symposium on Compiler Construction*, 6-10 August 1979, pp. 139-143. In *ACM SIGPLAN Notices* 14:8 (August 1979).

[77] LeBlanc, T. J. and R. P. Cook, "An Analysis of Language Models for High-Performance Communication in Local-Area Networks," *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, 27-29 June 1983, pp. 65-72. In *ACM SIGPLAN Notices* 18:6 (June 1983).

[78] Liskov, B., A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU," *CACM* 20 (August 1977), pp. 564-576.

[79] Liskov, B., "Primitives for Distributed Computing," *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, December 1979, pp. 33-42.

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

[81] Liskov, B. and M. Herlihy, "Issues in Process and Communication Structure for Distributed Programs," *Proceedings of the Third IEEE Symposium on Reliability in Distributed Software and Database Systems*, October 1983, pp. 123-132.

[82] Liskov, B. and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM TOPLAS* 5:3 (July 1983), pp. 381-404.

[83] Liskov, B., M. Herlihy, and L. Gilbert, "Limitations of Remote Procedure Call and Static Process Structure for Distributed Computing," Programming Methodology Group Memo 41, Laboratory for Computer Science, MIT, September 1984.

[84] Liskov, B., "Overview of the Argus Language and System," Programming Methodology Group Memo 40, Laboratory for Computer Science, MIT, February 1984.

[85] Lister, A., "The Problem of Nested Monitor Calls," *ACM Operating Systems Review* 11:3 (July 1977), pp. 5-7. Relevant correspondence appears in Volume 12, numbers 1, 2, 3, and 4.

[86] Lister, A. M. and K. J. Maynard, "An Implementation of Monitors," *Software — Practice and Experience* 6 (1976), pp. 377-385.

[87] Mao, T. W. and R. T. Yeh, "Communication Port: A Language Concept for Concurrent Programming," *IEEE Transactions on Software Engineering* SE-6:2 (March 1980), pp. 194-204.

[88] May, D., "OCCAM," *ACM SIGPLAN Notices* 18:4 (April 1983), pp. 69-79. Relevant correspondence appears in Volume 19, number 2 and Volume 18, number 11.

[89] Mitchell, J. G., W. Maybury, and R. Sweet, "Mesa Language Manual, version 5.0," CSL-79-3, Xerox Palo Alto Research Center, April 1979.

[90] Nelson, B. J., "Remote Procedure Call," Ph. D. Thesis, Technical Report CMU-CS-81-119, Carnegie-Mellon University, 1981.

[91] Parnas, D. L., "The Non-Problem of Nested Monitor Calls," *ACM Operating Systems Review* 12:1 (January 1978), pp. 12-14. Appears with a response by A. Lister.

[92] Powell, M. L. and B. P. Miller, "Process Migration in DEMOS/MP," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, 10-13 October 1983, pp. 110-118. In *ACM Operating Systems Review* 17:5.

[93] Pratt, T., *Programming Languages: Design and Implementation*, Prentice-Hall, Englewood Cliffs, 1975.

[94] Rashid, R. F. and G. G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel," *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, 14-16 December 1981, pp. 64-75.

[95] Robert, P. and J.-P. Verjus, "Toward Autonomous Descriptions of Synchronization Modules," pp. 981-986 in *Information Processing 77*, ed. B. Gilchrist, North-Holland, 1977. Proceedings of the 1977 IFIP Congress, Toronto, 8-12 August 1977.

[96] Roper, T. J. and C. J. Barter, "A Communicating Sequential Process Language and Implementation," *Software — Practice and Experience* 11 (1981), pp. 1215-1234.

[97] Saltzer, J. H., D. P. Reed, and D. D. Clark, "End-To-End Arguments in System Design," *ACM TOCS* 2:4 (November 1984), pp. 277-288.

[98] Scott, M. L., "Messages v. Remote Procedures is a False Dichotomy," *ACM SIGPLAN Notices* 18:5 (May 1983), pp. 57-62.

[99] Scott, M. L. and R. A. Finkel, "A Simple Mechanism for Type Security Across Compilation Units," Computer Sciences Technical Report #541, University of Wisconsin - Madison, May 1984.

[100] Scott, M. L. and R. A. Finkel, "LYNX: A Dynamic Distributed Programming Language," *Proceedings of the 1984 International Conference on Parallel Processing*, 21-24 August, 1984, pp. 395-401.

[101] Seitz, C. L., "The Cosmic Cube," *CACM* 28:1 (January 1985), pp. 22-33.

[102] Solomon, M. H. and R. A. Finkel, "The Roscoe Distributed Operating System," *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, December 1979, pp. 108-114.

[103] Spector, A. Z., "Performing Remote Operations Efficiently on a Local Computer Network," *CACM* 25:4 (April 1982), pp. 246-260.

[104] Stallings, W., "Local Networks," *ACM Computing Surveys* 16:1 (March 1984), pp. 3-41.

[105] Strom, R. E. and S. Yemini, "NIL: An Integrated Language and System for Distributed Programming," *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, 27-29 June 1983, pp. 73-82. In *ACM SIGPLAN Notices* 18:6 (June 1983).

[106] Tanenbaum, A. S., "A Tutorial on Algol 68," *ACM Computing Surveys* 8:2 (June 1976), p. 155.

[107] Tennent, R. D., "Another Look at Type Compatibility in Pascal," *Software — Practice and Experience* 8 (1978), pp. 429-437.

[108] United States Department of Defense, "Reference Manual for the Ada Programming Language," (ANSI/MIL-STD-1815A-1983), 17 February 1983.

[109] Walker, B., G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, 10-13 October 1983, pp. 49-70. In *ACM Operating Systems Review* 17:5.

[110] Weihl, W. and B. Liskov, "Specification and Implementation of Resilient, Atomic Data Types," *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, 27-29 June 1983, pp. 53-64. In *ACM SIGPLAN Notices* 18:6 (June 1983).