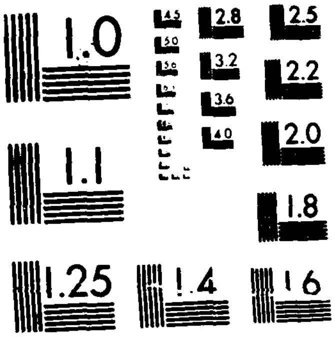
AD-A179 877   LANGUAGE SUPPORT FOR LOOSELY-COUPLED DISTRIBUTED   1/1
PROGRAMS(U) ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE
M L SCOTT SEP 86 TR-183 N00014-82-C-2087

UNCLASSIFIED   F/G 12/5   NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

(12)

Language Support for Loosely-Coupled Distributed Programs

Michael L. Scott
Department of Computer Science
The University of Rochester
Rochester, NY 14627

TR 183 (revised)
September 1986

rochester

Department of Computer Science
University of Rochester
Rochester, New York 14627

12

Language Support for Loosely-Coupled Distributed Programs

Michael L. Scott
Department of Computer Science
The University of Rochester
Rochester, NY 14627

TR 183 (revised)
September 1986

At the University of Wisconsin, this work was supported in part by NSF Grant MCS-8105904, DARPA Contract N00014-82-C-2087, and a Bell Telephone Laboratories Doctoral Scholarship. At the University of Rochester, the work is supported in part by NSF Grant DCR-8320136 and DARPA Contract DACA76-85-C-0001.

This paper will appear in the *IEEE Transactions on Software Engineering* in December 1986.

A179877

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
TR-183

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Language Support for Loosely- Coupled Distributed Programs

5. TYPE OF REPORT & PERIOD COVERED
Technical Report

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Michael L. Scott

8. CONTRACT OR GRANT NUMBER(s)
DACA76-85-C-0001 and
N00014-82-C-2087

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Computer Science Department
The University of Rochester
Rochester, New York 14627

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
DARPA/1400 Wilson Blvd.
Arlington, VA 22209

12. REPORT DATE
September 1986

13. NUMBER OF PAGES
36

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
Office of Naval Research
Information Systems
Arlington, VA 22217

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Distribution of this document is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES
None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
LYNX, Message Passing, Coroutines, Links, Abstraction, Distributed Resources

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A distributed operating system encourages a style of programming in which independently-developed processes interact in a non-trivial fashion at run time. Server processes, for example, must deal with clients that they do not understand, and certainly cannot trust. Interprocess communication can be written in a traditional, sequential language with direct calls to kernel primitives, but the result is both cumbersome and error-prone. Convenience and safety are offered by the many distributed languages proposed

20. ABSTRACT (Continued)

to date, but in a form too inflexible for anything other than the pieces of a single distributed program. A new language known as LYNX overcomes the disadvantages of both these previous approaches. Novel features of LYNX address problems encountered in the course of practical experience, writing distributed programs without high-level language support. Chief among these features are a virtual circuit abstraction called the link, and an unconventional coroutine mechanism that allows a server to maintain nested contexts for interleaved conversations with an arbitrary number of clients.

Language Support
for
Loosely-Coupled Distributed Programs

Michael L. Scott, member IEEE

July 1986

Abstract — A distributed operating system encourages a style of programming in which independently-developed processes interact in a non-trivial fashion at run time. Server processes, for example, must deal with clients that they do not understand, and certainly cannot trust. Interprocess communication can be written in a traditional, sequential language with direct calls to kernel primitives, but the result is both cumbersome and error-prone. Convenience and safety are offered by the many distributed languages proposed to date, but in a form too inflexible for anything other than the pieces of a single distributed program. A new language known as LYNX overcomes the disadvantages of both these previous approaches. Novel features of LYNX address problems encountered in the course of practical experience, writing distributed programs *without* high-level language support. Chief among these features are a virtual circuit abstraction called the **link**, and an unconventional coroutine mechanism that allows a server to maintain nested contexts for interleaved conversations with an arbitrary number of clients.

Index Terms — Coroutines, distributed computing, late binding, links, LYNX, message passing, process independence, programming languages.

I. Introduction

Advances in parallel architectures have spurred the development of a wide variety of distributed operating systems [6, 8, 30, 34, 35, 44]. Much of the functionality in these systems is provided *outside* the (replicated) kernel, by so-called system **servers**. Servers interact with users in precisely the same way that users interact with one another, making the distinction between application and system software increasingly unclear. A programming language for distributed computing must support safe, convenient communication for a dynamically-changing mix of loosely-coupled processes — processes designed in isolation, and compiled and loaded at disparate times.

The author is with the Department of Computer Science, University of Rochester, Rochester, NY 14627.

At the University of Wisconsin, this work was supported in part by NSF grant number MCS-8105904, DARPA contract number N0014-82-C-2087, and a Bell Telephone Laboratories Doctoral Scholarship. At the University of Rochester, the work is supported in part by NSF grant number DCR-8320136 and DARPA contract number DACA76-85-C-0001.

Under a distributed operating system, a process interacts with its environment through messages, much as a sequential process interacts with its environment through operations on files. It is tempting to presume that language features for message passing could reflect underlying communication primitives as easily as the I/O statements of a sequential language reflect underlying file operations. While such a direct mapping might be possible for processes whose interactions are limited to file-like operations, it is not possible for processes in general or for servers in particular. The extra complexity of IPC can be ascribed to several causes:

(1) Convenience and Safety

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 be able 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

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 trusted, passive entity whose behavior is determined by the operations performed upon it. A connection to an arbitrary process displays much more non-deterministic behavior.

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 (save perhaps for preparing the next file operation in high-performance, double-buffered applications), 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. Unfortunately, straightforward representation of remote operations will generally entail waiting for results to be returned. As described by Liskov, Herlihy, and Gilbert [28, 29], efficiency and clarity may best be realized with a dynamic set of tasks within a server, one for each uncompleted request.

Practical experience testifies to the significance of these issues. The Charlotte distributed operating system at the University of Wisconsin [14] is a case in point. Charlotte servers include 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. The original versions of these servers were written in a conventional sequential language with ordinary subroutine calls for access to the operating system kernel. As work progressed, serious problems arose. Those problems can be attributed directly to the issues just described:

- Programmers devoted a considerable amount of effort to packing and unpacking message buffers. The standard technique used type casts to overlay a record structure on an array of bytes. Program variables were assigned to or copied from appropriate fields of the record. The code was awkward at best and depended for correctness on programming conventions that were not enforced by the compiler. Errors due to incorrect interpretation of messages were 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 was not unusual for 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 was probably the most serious. In order to maximize concurrency and protect servers from recalcitrant clients, Charlotte programmers were forced to break the code that manages a conversation into many small pieces, separated by requests for communication. Servers would invoke the pieces individually so that conversations interleaved. Every Charlotte server developed 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 buried in the state information, obscured by the global loop. The program must save and restore that state in order to preserve the data structures associated with each conversation and in order to keep track of the current point of execution in what would ideally be straight-line code. Both tasks would be handled implicitly if each 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.

Previous research has addressed the complexity of IPC in several different ways. The problem of buffer management has been solved in several distributed systems by the development of so-called **stub** routines to pack and unpack parameters. A language-specific tool generates stubs automatically from interface descriptions. Birrell and Nelson's Lupine [4] and the Accent Matchmaker [23] are particularly worthy of note. The technique works best in languages that support procedures as first-class objects. Safety depends on integrating the stub generator into the compiler's type-checking mechanism and on preventing messages from being sent in any other way. If the language provides facilities for exception handling, then the second problem on the list above can be solved with stubs as well.

Addressing the third problem requires multiple cooperating threads of control in a single address space. Such threads are supported directly by the Amoeba distributed operating system [30], and may be realized through programming conventions in any operating system that allows processes to share memory. There is, however, a non-trivial cost associated with scheduling a server's tasks at the operating-system level, since creating a task or switching from one task to another requires a context switch into and out of the kernel. The designers of the Medusa distributed operating system [33] chose to implement coroutines at the user level rather than change the set of processes (**activities**) in a server (**task force**) at run time.

Though the first generation of Charlotte servers was indeed completed successfully, it became clear that direct use of system calls was an inadequate approach to writing systems programs. A stub generator was not an attractive alternative. Among other things, the language in which we were working (Modula-1 [13]) provided neither exceptions nor formal procedures, and its mechanism for blocking and unblocking threads of control was poorly suited for writing a message dispatcher. Moreover, no other, more suitable language was available on our machines. Since we were faced with the prospect of writing a compiler and run-time package in any event, we undertook to design a language that would overcome the disadvantages of the existing environment while sacrificing as few of its advantages as possible. In particular, we wished to obtain the benefits of high-level naming, type checking, exception handling, and automatic management of context while still allowing processes to be developed in mutual isolation, without the need for compiler-enforced global context.

Section II of this paper outlines the motivational factors that distinguish our work from that of previous language designers. Section III introduces a language we call LYNX. Rationale for the more important features of the language is provided in section IV, together with comparisons to previous research. The conclusion summarizes the significance of LYNX and discusses future plans.

II. Motivation

The complexity of interprocess communication has motivated the design of a large number of distributed programming languages. Work is still active on such languages and language tools as the Accent Matchmaker [23], Ada [43], Argus [27], CSP [21], EPL [6], Linda [18], NIL [40,41], SR [1,2], and Cedar [42] (with Nelson's RPC [4,31]). In the terminology of the previous section, most of the designs are convenient and safe. Their communication statements refer directly to program variables and they insist on type security for messages. Several provide special mechanisms for error handling and recovery. Most allow a process to be subdivided into more than one thread of control. None, however, appears to have been designed with independent processes in mind.

Return for a moment to the analogy between file operations and interprocess communication. Imagine a time-sharing system in which every data format in the file system must be declared in a database of types. Imagine that files are segregated according to the types they contain, and that consistency of access is enforced at compile time by checking programs against the database. If the file system is distributed across a local area network, imagine that the database must be kept consistent across machines. A file system along these lines could almost certainly be built, but its complexity and clumsiness hardly seem worth the security provided. Programmers routinely rely upon compile-time type checking for temporary files that are used in the course of a single run of a single program, but we suspect that they would balk at the need to do so for all files in all applications.

Similarly, we are inclined to doubt that compile-time knowledge of communication topology and types will be appropriate under all circumstances. From the point of view of a systems programmer, the principal disadvantage of existing distributed languages is a matter of *orientation*. Language designers have tended to think in terms of communication between the pieces of a *single* distributed program, rather than between processes that are really *separate* programs. The network of process interconnections, for example, must sometimes (as in CSP) be statically declared. Even if connections can be changed dynamically, it is generally necessary for any process that participates in introducing one process to another to understand the types of any messages a newly-created connection may carry. Client processes can always choose their servers, but servers are usually unable to distinguish between clients, to provide them with differing levels of service or extend to them differing levels of trust. No attempt is made to distinguish between local errors and remote errors, or to protect a process from the latter. Type checking is enforced by maintaining a global compiler name space. Even a trivial change to an interface will generally force the recompilation of every process that uses it.

For distributed systems software, a language must maintain the flexibility of explicit kernel calls while providing extensive features to handle errors, manage conversations, and make those calls convenient. A language that accomplishes these aims is introduced in the following section. The environment it provides is one in which programs can be pieced together quickly and easily from separately-developed processes, in the spirit of the well-known but substantially simpler **pipes** of UNIX.

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 a link can be moved from one process to another. 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. Type security is enforced on a message-by-message basis. Errors are deferred to exception handlers outside the normal flow of execution. Multiple conversations are supported by integrating the communication facilities with the mechanism for creating new threads of control.

III. LYNX Overview

Central to the philosophy of LYNX is the notion that processes are independent entities. Each can be written, compiled, linked, and loaded in total isolation, with no information whatsoever about any other process. The pieces of a single application, of course, will generally be written with their peers in mind, but it is the goal of LYNX to permit that mutual knowledge without requiring it for processes whose interactions are much less formal and structured.

Processes in LYNX execute in parallel, possibly on processors that share no common memory. Processes interact by sending messages on bi-directional communication **links**. Each process begins with an initial set of arguments, presumably containing at least one link to connect it to the rest of the world. Each link has a single process at each end. As an example of a simple application, consider a producer process that creates data of some type and sends that data to a consumer. Each process begins with a link to the other. The producer looks like this:

```
process 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.
```

The basic syntax and scope rules of LYNX are similar to those of Modula-2 [47]. Comments are defined as in Ada [43]. The word **entry** introduces a template for a remote operation. The general syntax is

entry opname (*in_args*) : *out_types* ;

In this case, the **transfer** entry has no reply parameters. The word **remote** indicates that the code for the operation is provided somewhere else.

The **connect** statement is used to request a remote operation. The vertical bar in the argument list separates request and reply parameters.

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

The current thread of control in the sending process is blocked until a reply message is received.

even if the list of reply parameters is empty. Our producer has only one thread of control (more complicated examples appear below), so in this case the process itself is blocked.

The consumer looks like this:

```
process 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.
```

The **accept** statement is used to serve an operation requested by the process at the other end of a link.

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

The **reply** statement returns its parameters to the process at the other end of linkname and unblocks the thread of control that requested the operation opname. The parameter types for opname must be defined by an entry declaration.

In keeping with the notion of process independence, neither the consumer nor the producer can name the other directly. Each refers only to the link that connects them. It is entirely possible that the consumer, having received all the data it wants, might pass its end of the link on to another process. Future requests for the transfer operation would be served by the new consumer. The producer would never know anything had happened.

A variable of type link accesses one end of a physical link, much as a pointer accesses an object in Pascal. Links are created by a built-in routine called newlink that returns references to a new pair of ends. New links are usually created for one of two reasons: either one end is to be passed to a newly-created process, or else both ends are to be passed to existing processes, to introduce each to the other. To make these common cases easier to write, newlink returns one of its results as a function value and the other as a reference parameter.

A producer/consumer pair could be created in LYNX with the following sequence of statements:

```
var L : link;
begin
  startprocess ("consumer", newlink (L));
  startprocess ("producer", L);
  ...
```

The strings "**consumer**" and "**producer**" serve to identify executable load images to the underlying operating system. The process that executes the **startprocess** statement may well be part of the same application as the processes it creates. Equally easily, it may be an operating-system process such as a command interpreter. In our implementation for the BBN Butterfly machine [3], a producer/consumer pair would be created in response to the following series of commands to the Butterfly **shell**:

```
[] link A B
[] xrun consumer @A
[] xrun producer @B
```

Since messages are addressed to links, not processes, it is not even necessary to connect the producer and consumer directly. An extra process could be interposed for the purpose of filtering or buffering the data. Neither the producer nor the consumer would know of the intermediary's existence.

```
[] link A B C D
[] xrun consumer @A
[] xrun intermediary @B @C
[] xrun producer @D
```

There is another way to write the consumer in LYNX. In the version above, the process contains a single thread of control that receives requests *explicitly*. We call the alternative *implicit* receipt. Instead of running a single thread in a loop, we can arrange for each appropriate request to create its own thread automatically. The code to be executed by a newly-created thread appears in the form of a **begin** ... **end** block for an entry, in place of the word **remote**.[1] The implicit-receipt version of our consumer looks like this:

[1] The header of the entry can still serve as the template for **connect** and **accept** statements; the word **remote** merely allows the code to be omitted when the current process does not provide the operation through implicit receipt.

```
process consumer (producer : link);

type data = whatever;

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

entry transfer (info : data);
begin
   reply;
   consume (info);
end transfer;

begin -- consumer
   bind producer to transfer;
end consumer.
```

The **reply** statement appears here in the body of an entry, without a matching **accept**. As with explicit receipt, it serves to unblock the thread of control that requested the current operation. The producer shown above can be used with either version of the consumer, without modification.

The **bind** statement serves to define an "appropriate" request.

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

Each of the entries mentioned in a bind statement must have a **begin ... end** block. A subsequent request on one of the mentioned link ends for one of the mentioned operations will create a new thread to execute the matching entry. Bindings can also be broken:

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

The ability to make and break bindings at run time is a powerful mechanism for access control, as we shall see below.

Entries may be declared at any level of lexical nesting. Non-global data may therefore be shared by more than one thread of control. A newly-created thread begins execution in the naming environment of the **bind** statement that permitted its creation. The activation records accessible at any given time will form a tree (a **cactus stack** [20]), with a separate thread corresponding to each leaf. From the point of view of any one thread, the path back to the root looks like a normal stack. To simplify reclamation of stack frames, a thread is not allowed to leave a given scope until any descendant threads still active in that scope have completed.

A **reply** statement can occur anywhere inside an entry; the thread that executes it continues to exist until it reaches the end of its code. Often a newly-created thread will allocate new data structures, create some bindings for nested threads, send a reply to indicate that it is ready, and then continue to receive related requests throughout a lengthy conversation. The file server example in section III.D will contain such a thread for each of its open files.

The threads of control within a single process do *not* execute in parallel; each process continues to execute a single thread until it blocks.[2] The process then takes up some other thread where it last left off. If no thread is runnable, then 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.

Though the implicit-receipt version of the consumer will contain a thread for every invocation of the transfer operation, it is likely that only one such thread will exist at a time. For a slightly more complicated example, consider the buffer process mentioned above. Interposed between a producer and consumer, the buffer serves to smooth out fluctuations in their relative rates of speed.

```
process 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 % size + 1;
   reply;
end transfer;
```

[2] The circumstances under which a thread may block are defined in section III.A

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

Here the header of the **transfer** entry serves to define the structure of both incoming and outgoing **transfer** requests. The **await** statement blocks the current thread until the specified condition is true. There is no need to worry about simultaneous access to **buf**. **firstfree**. or **lastfree**. because the coroutine semantics guarantee that only one thread can execute at a time.

A. Execution Details

Every LYNX process begins with a single thread of control, executing the process's main **begin ... end** block. New threads are created in response to incoming requests on links bound to entries. and may also be created explicitly by executing a **call** (fork) statement in an existing thread.

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

As with **connect**. the calling thread is blocked until the entry replies.

Context switches between threads happen only 1) at **connect**. **accept**. **reply**. and **call** statements, 2) at **await** statements. and 3) when the current thread reaches the end of a scope in which descendant threads are still active or in which bindings exist that might cause the creation of descendant threads.

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. The **startprocess** statement, for example, might be implemented by sending a request to a process manager written in LYNX. Each such request might create a new thread of control within that manager. Separate threads could share the same link between the process manager and file server. Their requests to open and read executable files would interleave transparently.

When all of a process's threads are blocked, run-time support routines attempt to receive a message on any of the links for which there are outstanding **accepts** or bindings, or on which replies are expected for outstanding **connects**. Incoming replies can only have been sent in response to an outgoing request. Each such reply can therefore be delivered to an appropriate thread of control. Incoming replies, by contrast, can be unexpected or unwanted. Competing goals come into play. On the one hand, the implementation should detect (and reject) requests for invalid or bogus operations. On the other hand, it should distinguish such cases from requests for operations for which a server is not yet ready, but will be sometime "soon." LYNX addresses these concerns by defining a valid request to be one for which the server *will* be ready when all its threads are blocked.

Incoming messages are not examined until all threads are blocked. The operation name of a request is compared against those of the outstanding **accepts** and bindings for its link. If a match is found, then an appropriate thread can be made ready and execution can continue. If there are no **accepts** or bindings, then consideration of the message is postponed. If **accepts** or bindings exist, but none of them match the request, then the message is discarded and an INVALID_OPERATION exception is raised in the thread that executed the **connect** statement at the other end of the link. Exceptions are discussed in more detail in section III.D.

One consequence of the above rules is that there is no way in LYNX to receive a message asynchronously. Real-time device control cannot be programmed, nor can any algorithm in which incoming messages must *interrupt* the execution of lower-priority "background" computation. There are currently no plans to accommodate LYNX to hard, real-time constraints. For less demanding applications, a low-cost polling function can be used by a background thread to relinquish control when higher-priority messages arrive. The built-in function idle returns false whenever the communication for which another thread is waiting has completed. Otherwise it returns true. We have used the idle function in a distributed game-playing program based on Fishburn's algorithms for alpha-beta search [17]. Threads that are evaluating pieces of the game tree execute the statement

```
    await idle;
```

at the top of an outer loop. Messages containing updated alpha-beta values (for better game-tree pruning) are therefore received within a reasonable amount of time. Evaluation of idle is fast enough that performance does not suffer.

B. Link Movement

Much of the power of LYNX derives from the ability to move the ends of links. Language semantics specify that every link end is accessible to only one process at a time. If a data structure containing one or more link variables is enclosed in a message, then the transmission of that message will have have the side effect of *moving* the referenced link ends 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 can be used to access the green end of L, but X is a dangling reference. Loosely speaking, the sender of a link variable loses access to the end of the link involved.

We have seen (in the producer/consumer example) how moving links are used to establish connections between newly-created processes. They can be used at other times as well. A link between a server and a client can be passed on to a new client when the first one doesn't need it any more. It can also be passed on to a new *server* (functionally equivalent to the old one, presumably) in order to balance work load or otherwise improve performance. A *name server* process can even keep a database of server names and links. Clients in need of a particular service can ask the name server for a link on which to request that service.

To facilitate use of a name server, we have established a convention for introducing a new client to a server that can support more than one client at a time. Each such server binds its name-server link to a **newclient** entry.

```
entry newclient (client : link);
```

When asked for a link to, say, a mail server, the name server creates a new link, passes one end to the mail server in a request for the **newclient** operation, and returns the other end to the client. In the **newclient** entry, the mail server binds the newly-received client link to some "standard" set of entries.

C. Access Control

Unlike most distributed languages, LYNX allows a server to control precisely which clients have access to the operations it provides. By making and breaking bindings at run time, a process can enforce a simple and highly effective form of access control. Consider, for example, the famous readers/writers problem [11]. A server controls a resource that behaves like a large collection of data. Two operations are provided: reading and writing individual data items. More than one client may read at once, but for the sake of consistency a writer requires exclusive access to the entire data structure. Each client performs its operations in the course of read or read/write *sessions.* It begins a session by requesting permission to read or write. It ends a session by informing the server that it is through. Each process is guaranteed that no one else will perform a write operation while it is in the middle of its current session. Each writer is also guaranteed that no one else will perform a read operation. Most important, the decision as to which data to read or write is allowed to depend on the results of previous read operations; a client need not know exactly what it wants to read or write at the time it begins its session.

Here is a solution in LYNX:

```
process readwrite (firstclient : link);

var readers, writers : 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
  await writers = 0;
  readers +:= 1;
  unbind curlink from startwrite, startread;
  bind curlink to doread, endread;
  reply;
end startread;

entry startwrite;
begin
  await readers = 0 and writers = 0;
  writers +:= 1;
  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 -:= 1;
  reply;
end endread;
```

```
entry endwrite;
begin
   unbind curlink from doread, dowrite, endwrite;
   bind curlink to startread, startwrite;
   writers -:= 1;
   reply;
end endwrite;

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

begin -- initialization
   readers := 0; writers := 0;
   call newclient (firstclient |);
end readwrite.
```

To simplify presentation, we have not worried in this example about starvation of either readers or writers. A more careful solution would require some 30 lines of additional code (and would in almost any language). Global variables would keep track of how many readers and writers were waiting to get access. **Await** statements would be modified to take these variables into account. **Startread** would block when there were waiting writers. **Endwrite** would unblock waiting readers first.

By making and breaking bindings, the server is able to ensure that clients are physically unable to perform operations for which they have not obtained authorization. The built-in function **curlink** returns a reference to the link on which the request message arrived for the closest lexically-enclosing entry (in this case the current entry). From the point of view of a reader, a typical session would look something like this:

```
connect startread on RWlink;
-- series of (possibly interrelated) doread requests
connect endread on RWlink;
```

A read/write session looks like this:

```
connect startwrite on RWlink;
-- series of (possibly interrelated) doread and dowrite requests
connect endwrite on RWlink;
```

Any client that requests read or write operations without obtaining permission, or that requests a startread, startwrite, endread, or endwrite operation out of order will feel an INVALID_OPERATION exception.

The **newclient** convention has been used in this example. We have written the server to take a single initial argument: a link to a single client. Additional clients are introduced by invocations of **newclient** over links from existing clients. We have used the **call** statement to establish the same bindings for the initial client as are established for late-comers.

D. Exceptions

LYNX provides an exception handling mechanism to 1) cope with exceptional conditions that arise in the course of message passing, and 2) allow one thread to interrupt another. Exception handlers may be attached to any **begin** ... **end** block. Such blocks comprise the bodies of procedures, entries, processes, 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 a link on which a thread is trying to send or receive.

All of a process's links are destroyed when it terminates or crashes. Additional exceptions can be defined by the programmer.

A built-in exception is raised in the current thread of control when one of the above conditions prevents that thread's normal continuation. Both built-in and user-defined exceptions can also appear in an explicit **raise** statement. In either case, the search for an appropriate handler begins in the current block. If that block has no handler, the exception is raised in the next enclosing block, or in the previous scope on the dynamic chain if the block is a procedure or function. Propagation halts at the scope in which the thread began. If the exception is not handled at that level, then 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 requesting process as well. If the propagation escapes a scope in which nested threads are still active, those threads are aborted recursively.

In addition to the **raise** statement, LYNX provides an **announce** statement to allow one thread to interrupt another. An **announced** 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 handlers refer to them by name, **announced** exceptions must be declared in a scope visible to all the threads that use them. The coroutine semantics guarantee that threads feel exceptions only when blocked.

Announced exceptions are useful for protocols in which one thread may discover that the communication for which another thread is waiting is no longer appropriate (or possible). One example is found in a stream-based file server. The code below sketches the form that such a server might take.

```
process fileserver (switchboard : link);
type string = whatever; bytes = whatever;

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

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

    entry writeseek (newptr : integer);
    begin
        writeptr := newptr; reply;
    end writeseek;

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

    entry readseek (newptr : integer);
    begin
        readptr := newptr; announce seeking; reply;
    end readseek;

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

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

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

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

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

Like the **readwrite** server, the file server begins with a single initial link. Here we have assumed that the link is attached to a name server (the **switchboard**), and not to an ordinary client. When the switchboard sends the file server a link to a new client (line 48), 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 this example (lines 3-47).

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 23), each thread creates a new link (line 24) 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 [5], bulk data transfers are initiated by the producer (with **connect**) and **accepted** by the consumer. As we have seen, this asymmetry allows the transparent insertion of an intermediate filter or buffer. 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 20, caught at line 36) in the file-server thread that is attempting to send data out over the link.

Clients close their files by destroying the corresponding links.[3] A thread that tries to use a destroyed link feels a REMOTE_DESTROYED exception (caught at line 38 in the file server). Bindings for a destroyed link are broken automatically. These mechanisms suffice in this example to clean up the context for a file.

IV. Discussion

Every language is heavily influenced by the perspective of its designer(s). Existing distributed languages grew out of efforts to generalize sequential languages, first to multiple processes, then to multiple processors. LYNX evolved in the opposite direction. It *began* with the distributed processes and worked to increase their effectiveness through high-level language support.

Aiming for elegance, previous languages attempted to invent a small set of fundamental concepts. Many of their decisions would be inappropriate for the style of programming to which we had grown accustomed under Charlotte. Resources, for example, are often confused with either processes or operations. CSP and EPL support remote naming at the level of an entire process only. They fail to recognize that a process may implement an *arbitrary number of resources*. NIL and SR (and in some sense Ada, Argus, and Cedar as well) provide capability variables that permit naming at the level of individual operations, but since a resource may support more than one operation, these capabilities must be packaged up in records. Only SR allows a server to provide separate instances of the same operation for separate resources.[4] Similar confusion between processes and threads has caused designers to forbid the existence of multiple threads (as in CSP and NIL), or to specify that threads may execute in parallel (as in Ada, Argus, Cedar, EPL, Linda, and SR). The first extreme complicates the management of context; the second requires special mechanisms to protect shared data.

That existing distributed languages should be inappropriate for the server programs of a particular operating system should not be especially surprising. These languages have, for the most part, been designed to address the following question: "Here is an important application; how can we run it on multiple machines?" LYNX attempts to address the complementary question: "Here are some programs already in use on multiple machines; how can we impose some structure on

[3] **Destroy** is a built-in procedure that takes a single parameter of type link. Variables accessing either end of a destroyed link become dangling references.

[4] N. B.: SR terminology differs from that used here. What we call a process is called a resource in SR. What we call a thread is called a process. Our notion of resource has no direct analog.

their interactions?"

A distributed operating system like Charlotte (or like any of the others in the references) can be used for embedded applications. It can also be used for a dynamically-changing mix of smaller applications, in the style of conventional time-sharing. LYNX is based on the assumption that largely-unrelated processes may need to communicate from time to time, and that high-level language support can make that communication both safer and more convenient.

The notion of process independence is to a large extent the legacy of UNIX [36]. It is certainly not the only way to go about building software, but it is one that has proven highly successful for sequential computation and that merits consideration for parallel environments as well. Much of the power and popularity of UNIX derives from the ability to piece together applications from a large collection of small but general tools. LYNX maintains this level of flexibility while providing mechanisms to manage the extra complexity of a parallel environment. Chief among these mechanisms are the link and the thread of control. Links support interaction between processes; threads of control support management of context within processes.

A. 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 thought of 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 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 [47], Ada packages [43], and Clu clusters [26] 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 helps to ensure that a resource and its users do not 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 incorrect 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, for example, 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 can 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 an 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 readers/writers example of section III.C, 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 [46].[5]

[5] The "solution" in reference 22 (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

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 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 link movement would depend on which end was enclosed; special rules would apply to the movement of ends attached to clients. 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.[6]

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.[7]

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

[7] We have used a technique based on hashing to minimize the cost of run-time checks [38]. The expense per message is less than 10 microseconds

Several existing languages provide late binding for communication paths. Ada. Argus. Cedar. and SR provide variables that hold a reference to a process. NIL and SR provide variables that hold a reference to a single operation. 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 link can change roles dynamically without forcing a server to deal explicitly with inappropriate requests.

LYNX type checking also differs from that of previous languages in its use of structural equivalence ([19], p. 92). The alternative, name equivalence, requires the compiler to maintain a global name space for types. 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. 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, 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.

Dynamic checking has been used in conjunction with a global name space for types in EPL. The Eden designers call this method **abstract typing** [6]. The compiler verifies that each request for a remote operation agrees with the declaration of that operation in the name space. Only when the request occurs at run time, however, does EPL check to see that the requestor and provider of an operation were compiled with the same declaration. LYNX differs from this approach only in that it uses the structure of message parameters, rather than the globally-unique location of a declaration, as the basis of type compatibility. Such errors as two requests in the same process for the same operation but with different parameter types are still caught at compile time.

Probably the most serious problem with run-time checking is that programming errors that would have been caught at compile time in other languages may not be noticed until significantly later in LYNX or EPL. We have accepted this cost in LYNX as the price of flexibility. As a practical matter, we tend to follow the Eden style, compiling individual processes on the basis of shared files of declarations. Though type clashes can in principle be announced at run time, it seldom happens in practice.

A second, serious cost of the LYNX approach to types is the less-than-perfect checking implied by structural equivalence. Variables with the same arrangement of components will be accepted as compatible even if the abstract meanings of those components are completely unrelated. This cost, too, we have been willing to accept, with the understanding that no type system, no matter how exacting, will *ensure* that messages are meaningful. The goal of type checking is to reduce the likelihood of data misinterpretation, not to eliminate it altogether.

B. 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. Such relatively early languages as Algol-68, PL/I, and SIMULA allow more than one thread to operate inside a single module and share that module's data. The threads are 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, Cedar, EPL, and SR, a resource is an isolated module. Argus calls such modules **guardians**; Cedar calls them **modules**, EPL calls them **objects**, and SR calls them **resources**. Each module is inhabited by one or more **processes**. Semantics specify that the processes execute in parallel, but implementation considerations prevent their assignment to machines that share no memory. In effect, the "processes" of these other languages are the threads of control of LYNX. Guardians, modules, objects, and resources correspond to LYNX processes.

Ada allows data to be shared by arbitrary processes (**tasks**) that execute in parallel. It has no notion of modules that are *inherently* disjoint. In the absence of a shared-memory architecture, 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. The former option is facilitated by semantics that require consistency for shared data only when tasks are synchronized.

While simulated parallelism may be aesthetically pleasing, it does not reflect the nature of most 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 EPL and Cedar, 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, Linda, 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 provides the appearance of serial execution for even large-grain operations. Programmers have complete control over the exact meaning of atomicity for individual data types [45]. 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 [7]. 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. Similar semantics are provided by the Amoeba distributed operating system [30], where each process is composed of a set of tasks that share data but execute in mutual exclusion.

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) Requests can be received explicitly (with **accept**), as well as implicitly (through bindings).

(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, procedures, and entries 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 necessary tree of activation records are well understood [20]. Activation records for any subroutine that might not return before the next context switch must be allocated from a heap. Allocators for this purpose have been built before [25], with excellent performance.

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.

C. 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*. Rationale for providing both options is discussed in detail elsewhere [37]. The gist of the argument is that each approach has applications for which it is appropriate and others for which it is both awkward and confusing.

Implicit receipt reflects the externally-driven nature of most servers. It recognizes that many processes are essentially *passive*, sitting idle until called from outside. With implicit receipt, the programmer can allow servers to converse with arbitrary numbers of clients without guessing how many threads to allocate ahead of time and without replicating code in every server to create new threads dynamically.

Explicit receipt is most useful for the exchange of messages between active, cooperating peers. Its use was demonstrated by the producer and consumer of section III.

Some existing languages, notably StarMod [9, 10], 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.

D. Experience

An implementation of LYNX for Charlotte has been in use since 1984. It runs on the University of Wisconsin's Crystal multicomputer [12]. A second, paper design was created for the SODA distributed operating system designed by Jonathan Kepecs [24]. A third implementation is now in use at the University of Rochester, where it runs on the BBN Butterfly Parallel Processor [3]. Details can be found in reference 39.

At Wisconsin, the standard Charlotte servers were originally written in Modula ([13], sequential features only) with direct calls to the IPC primitives of the kernel. Many of those servers have now been written in LYNX. Several 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 was 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.[8]
- 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.
- LYNX can be implemented at acceptable cost. For Charlotte, the overhead of the language run-time package added less than ten percent to the transmission times for messages (while simultaneously adding a significant amount of functionality). On the Butterfly, simple remote operations complete in just over two milliseconds. Code tuning and protocol optimizations now under development are likely to improve this figure by 30 to 40%. In several cases, re-implementation of a server in LYNX has led to significantly *faster* code, because programmers are no longer tempted to simplify their task by waiting for the completion of individual communication requests.

[8] Object code from LYNX tends to be about 50% larger than its sequential counterpart. The difference can be attributed to 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 calls. In addition, every LYNX program is linked to a substantial amount of run-time support code: the message dispatcher, communication routines, and code to manage exceptions and threads.

V. Conclusion

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 remote processes

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

Even for the pieces of a single distributed program, 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. By relying on structural type equivalence for messages, LYNX avoids unnecessary recompilations when definitions change.

Support for tightly-coupled programs, however, is not central to the goals of LYNX. The real significance of the language is in areas outside the focus of previous research. LYNX supports applications for which other languages were never intended. It adapts the advantages of a high-level language to processes designed in nearly total isolation.

Ongoing work with LYNX is focused on two fronts: mechanisms and implementation. For the former, researchers at both Wisconsin and Rochester are working to evaluate the language through practical experience [15, 16]. Several enhancements have already been suggested:

- A *cobegin* construct may be offered as an additional means of creating new threads of control. 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.

- For the Butterfly, mechanisms may be added to take more direct advantage of the shared-memory architecture. It is currently possible for two processes to obtain pointers (from the operating system) to a shared block of Butterfly memory. Communication over links can then be used to synchronize access. Changes to the language might support this sharing in a more safe and structured way. Alternatively, the semantics of mutual exclusion of threads might be relaxed in favor of parallel execution. Such a change would represent a significant departure from the philosophy of section IV.B, but might be of use in a number of emerging hardware configurations, including networks of multiprocessor workstations. Finally, it might be possible to design a compiler that would permit non-interfering threads to execute in parallel without changing the language semantics. It is unclear exactly how much parallelism could be exploited in this fashion. The prospect is reminiscent of past attempts to discover parallelism in ordinary sequential languages [32], and may be ill-advised.

- Farther down the road, the entire notion of links might be removed from the language itself and placed under user control. There is some reason to be skeptical of any "systems" language that requires "a fixed, hidden, and large so-called run-time package [48]." With suitable facilities for data and control abstraction, such IPC facilities as **connect**, **accept**, and **bind** might be provided by library routines. We have been pleased by the effectiveness of links, but have no illusions that they are the only useful abstraction for distributed computing. In the context of work on the Butterfly we have begun to investigate the extent to which a wide variety of programming models, from pure shared memory through connection-less message passing, might be made to coexist within a single, common framework for interprocess interaction. Such a goal would be facilitated by a language in which users could choose the model most appropriate for the application at hand.

The research on implementation techniques is particularly concerned with the speed of message passing. We are experimenting with novel data structures and algorithms to enhance the efficiency of common communication patterns. In addition, we are exploring the relationship between efficiency and the level of abstraction of kernel primitives. Preliminary comparisons among the Charlotte/Crystal, SODA, and Butterfly implementations suggest that efficiency is best achieved with a comparatively low-level interface between the language and the operating system [39]. Through extensive profiling and examination of code paths, we hope to obtain a detailed analysis of message overhead and of the inherent limits on its speed.

The design of LYNX was an exercise in practical problem-solving. The language must therefore 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.

Acknowledgments

Much of the research described in this article was conducted in the course of doctoral studies at the University of Wisconsin under the supervision of Associate Professor Raphael Finkel. Critical comments from the referees led to significant improvements over an earlier draft.

References

[1] G. R. Andrews, "The Distributed Programming Language SR — Mechanisms, Design and Implementation," *Software — Practice and Experience*, vol. 12, pp. 719-753, 1982.

[2] G. R. Andrews and R. A. Olsson, "The Evolution of the SR Language," TR 85-22, Department of Computer Science, The University of Arizona, 14 October 1985.

[3] BBN Laboratories, "Butterfly® Parallel Processor Overview," Report #6148, Version 1, Cambridge, MA, 6 March 1986.

[4] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM TOCS*, vol. 2, no. 1, pp. 39-59, February 1984. Originally presented at the *Ninth ACM Symposium on Operating Systems Principles*, 10-13 October 1983.

[5] A. P. Black, "An Asymmetric Stream Communication System," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pp. 4-10, 10-13 October 1983. In *ACM Operating Systems Review*, vol. 17, no. 5.

[6] A. P. Black, "Supporting Distributed Applications: Experience with Eden," *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pp. 181-193, 1-4 December 1985.

[7] P. Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept," *CACM*, vol. 21, no. 11, pp. 934-941, November 1978.

[8] D. R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pp. 129-140, 10-13 October 1983. In *ACM Operating Systems Review*, vol. 17, no. 5.

[9] R. P. Cook, "*Mod – A Language for Distributed Programming," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 6, pp. 563-571, November 1980.

[10] R. P. Cook, "The StarMod Distributed Programming System," *IEEE COMPCON Fall 1980*, pp. 729-735, September 1980.

[11] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent Control with 'Readers' and 'Writers'," *CACM*, vol. 14, no. 10, pp. 667-668, October 1971.

[12] D. J. DeWitt, R. Finkel, and M. Solomon, "The CRYSTAL Multicomputer: Design and Implementation Experience," Computer Sciences Technical Report #553, University of Wisconsin – Madison, September 1984.

[13] R. Finkel, 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.

[14] R. Finkel, 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.

[15] R. Finkel, A. P. Anantharaman, S. Dasgupta, T. S. Goradia, P. Kaikini, C.-P. Ng, M. Subbarao, G. A. Venkatesh, S. Verma, and K. A. Vora, "Experience with Crystal, Charlotte, and LYNX," Computer Sciences Technical Report #630, University of Wisconsin – Madison, February 1986.

[16] R. Finkel, B. Barzideh, C. W. Bhide, M.-O. Lam, D. Nelson, R. Polisetty, S. Rajaraman, I. Steinberg, and G. A. Venkatesh, "Experience with Crystal, Charlotte, and LYNX: Second Report," Computer Sciences Technical Report #649, University of Wisconsin – Madison, July 1986.

[17] J. P. Fishburn, "An Analysis of Speedup in Parallel Algorithms," Ph. D. thesis, Computer Sciences Technical Report #431, University of Wisconsin – Madison, May 1981.

[18] D. Gelernter. "Generative Communication in Linda." *ACM TOPLAS*. vol. 7. no. 1. pp. 80-112. January 1985.

[19] C. Ghezzi and M. Jazayeri. *Programming Language Concepts*. New York: John Wiley and Sons. 1982.

[20] E. A. Hauck and B. A. Dent. "Burroughs' B6500/B7500 Stack Mechanism." *Proceedings of the AFIPS Spring Joint Computer Conference*. vol. 32. pp. 245-251. 1968. Chapter 16. pp. 244-250. in *Computer Structures: Principles and Examples*. by D. P. Siewiorek. C. G. Bell. and A. Newell. New York: McGraw-Hill. 1982.

[21] C. A. R. Hoare. "Communicating Sequential Processes." *CACM*. vol. 21. no. 8. pp. 666-677. August 1978.

[22] J. D. Ichbiah. 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*. vol. 14. no. 6. June 1979.

[23] M. B. Jones. R. F. Rashid. and M. R. Thompson. "Matchmaker: An Interface Specification Language for Distributed Processing." *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*. pp. 225-235. January 1985.

[24] J. Kepecs and M. Solomon. "SODA: A Simplified Operating System for Distributed Applications." *ACM Operating Systems Review*. vol. 19. no. 4. pp. 45-56. October 1985. Originally presented at the *Third ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing*. 27-29 August 1984.

[25] B. W. Lampson and D. D. Redell. "Experience with Processes and Monitors in Mesa." *CACM*. vol. 23, no. 2, pp. 105-117. February 1980.

[26] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. "Abstraction Mechanisms in CLU." *CACM*, vol. 20, pp. 564-576. August 1977.

[27] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM TOPLAS*, vol. 5, no. 3, pp. 381-404, July 1983.

[28] B. Liskov 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*, pp. 123-132, October 1983.

[29] B. Liskov, 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, revised October 1985.

[30] S. J. Mullender and A. S. Tanenbaum, "The Design of a Capability-Based Distributed Operating System," Report CS-R8418, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, 1984.

[31] B. J. Nelson, "Remote Procedure Call," Ph. D. Thesis, Technical Report CMU-CS-81-119, Carnegie-Mellon University, 1981.

[32] A. Nicolau, "Uniform Parallelism Exploitation in Ordinary Programs," *Proceedings of the 1985 International Conference on Parallel Processing*, pp. 614-618, 20-23 August 1985.

[33] J. D. Ousterhout, D. A. Scelza, and S. S. Pradeep, "Medusa: An Experiment in Distributed Operating System Structure," *CACM*, vol. 23, no. 2, pp. 92-104, February 1980.

[34] M. L. Powell and B. P. Miller, "Process Migration in DEMOS/MP," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pp. 110-118, 10-13 October 1983. In *ACM Operating Systems Review*, vol. 17, no. 5.

[35] R. F. Rashid and G. G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel," *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pp. 64-75, 14-16 December 1981.

[36] D. M. Ritchie and K. Thompson, "The UNIX Time Sharing System," *CACM*, vol. 17, no. 7, pp. 365-375, July 1974.

[37] M. L. Scott, "Messages v. Remote Procedures is a False Dichotomy," *ACM SIGPLAN Notices*, vol. 18, no. 5, pp. 57-62, May 1983.

[38] M. L. Scott and R. A. Finkel, "A Simple Mechanism for Type Security Across Compilation Units," Computer Sciences Technical Report #541, University of Wisconsin – Madison, May 1984. Revised version to appear in *IEEE Transactions on Software Engineering*.

[39] M. L. Scott, "The Interface Between Distributed Operating System and High-Level Programming Language," *Proceedings of the 1986 International Conference on Parallel Processing*, 19-22 August 1986.

[40] R. E. Strom 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*, pp. 73-82, 27-29 June 1983. In *ACM SIGPLAN Notices*, vol. 18, no. 6.

[41] R. E. Strom and S. Yemini, "The NIL Distributed Systems Programming Language: A Status Report," *ACM SIGPLAN Notices*, vol. 20, no. 5, pp. 36-44, May 1985.

[42] D. C. Swinehart, P. T. Zellweger, and R. B. Hagmann, "The Structure of Cedar," *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pp. 230-244, 25-28 June 1985. In *ACM SIGPLAN Notices*, vol. 20, no. 7, July 1985.

[43] United States Department of Defense, "Reference Manual for the Ada® Programming Language," (ANSI/MIL-STD-1815A-1983), 17 February 1983.

[44] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pp. 49-70, 10-13 October 1983. In *ACM Operating Systems Review*, vol. 17, no. 5.

[45] W. Weihl and B. Liskov, "Specification and Implementation of Resilient, Atomic Data Types," *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, pp. 53-64, 27-29 June 1983. In *ACM SIGPLAN Notices*, vol. 18, no. 6.

[46] J. Welsh and A. Lister, "A Comparative Study of Task Communication in Ada," *Software — Practice and Experience*, vol. 11, pp. 257-290, 1981.

[47] N. Wirth, *Programming in Modula-2*, Third, Corrected Edition. Texts and Monographs in Computer Science, ed. D. Gries, Berlin: Springer-Verlag, 1985.

[48] N. Wirth, "From Programming Language Design to Computer Construction," *CACM*, vol. 28, no. 2, pp. 159-164, February 1985. The 1984 Turing Award Lecture.

END

6-87

DTIC