Proc. 1984 International Conf. Parallel Processing
pp 395-401

LYNX: A DYNAMIC DISTRIBUTED PROGRAMMING LANGUAGE

Michael L. Scott
Raphael A. Finkel

Department of Computer Sciences
University of Wisconsin - Madison
1210 W. Dayton
Madison, WI 53706

Abstract -- This paper describes a new language for multi-computer systems programming. Processes in the language communicate by sending messages over communication channels called links. Links may be created, manipulated, and destroyed dynamically to provide complete run-time control over the interconnections among processes. Message type checking is performed at run time. Messages may be received explicitly or may trigger the execution of entry procedures. A control-flow mechanism similar to coroutines simplifies the ordering of entry procedures.

Introduction

Charlotte [1] is a distributed operating system under development at the University of Wisconsin at Madison. Charlotte runs on a collection of VAX 11/750 processors connected by a high-speed token ring. The Charlotte kernel is replicated on each machine. It provides two principal abstractions: processes and links. Processes do not share data. Their only means of interaction is through the exchange of messages on links. A link is a duplex communication channel between a pair of processes. Links can be created and destroyed, and their ends can be moved from one process to another. Most traditional operating system functions are provided by server processes that run on top of the kernel. This paper discusses LYNX, a language for writing such servers.

Motivation

There are two principal characteristics of operating system server processes that have complicated our search for an appropriate language:

(1) Servers must deal with an intricate web of interconnections among user processes and other servers. This web changes constantly as processes come and go. A programming language for servers must allow processes to examine and manipulate their interconnections. It must provide strict type checking on messages, yet allow the same communication channel to be used for multiple message types.

(2) At a given time, a typical server will be in the process of handling several different and largely independent requests in various stages of completion. For the sake of clarity, a programming language for servers must allow these requests to be handled by separate threads of control, without incurring the overhead of operating system support for each individual thread. It must also allow new threads of control to be created implicitly, in response to new requests, without requiring an explicit search for incoming messages.

To our knowledge, no existing distributed language meets the above requirements. We have set out to design one that does.

This paper is divided into two main sections. The first contains an informal language description. The second discusses the language and compares it with previous proposals. The Conclusion is followed by a brief description of the current status of our work.

Language Description

Main Concepts

The three most important concepts in LYNX are the module, the entry, and the link. A module encapsulates data, procedures, and entries. Modules may nest. An entry is a block of code resembling a procedure, but designed to be executed as an independent thread of control. A link is a two-way channel for messages.

A module is a syntactic structuring tool. It is an abstraction of a single node of a multicomputer. Each outermost module is implemented by a separate process, provided by the operating system. Separate modules execute concurrently.

Each process begins with a single thread of control, executing the initialization code of its outermost module. It can create new threads by calling entries or by arranging for them to be called automatically, in response to incoming messages. Separate threads do not execute concurrently; a given process continues to execute a given thread until it blocks. It then takes up some other thread where it last left off. If no

This work was supported in part by NSF grant number MCS-8105904, by Arpa contract number N0014/82/C/2087, and by a Bell Telephone Laboratories Doctoral Scholarship.

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 commands are discussed in a later section.

Lexical scope in LYNX is defined as in Modula [2]. There are no restrictions on the nesting of modules, procedures, and entries. Non-global data may therefore be shared by more than one thread of control. The activation records accessible at any given time will form a tree, with a separate thread of control 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 been completed.

The sequential features of LYNX are Algol-like. We will not discuss them here.

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 inter-process interaction. The language provides a primitive type called "link." A link variable accesses one end of a link, much like a pointer in Pascal [3]. 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.

Ends of links may be sent in messages. 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 that is bound to a entry (see below), or on which there are outstanding sends or receives.

Sending Messages

Message transmission looks like a remote invocation:

```
connect opname ( <expr list> ‖ <var list> )
    on linkname ;
```

Run-time support routines package the 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.

Receiving Messages Explicitly

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

```
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 replys.

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 must be 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 links to entries:

```
bind <link list> to <entry list> ;
```

After binding, an incoming message on any of the mentioned links 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 the reply statement (without a matching accept).

A link may be bound to more than one entry. The bindings need not be created at the same time. A bound link 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 a process's threads are blocked, the run-time support package attempts 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 <procedure 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>) ;

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.

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 [4] or PL/I [5].

Exception handlers may be inserted at the end of any begin ... end block. Such blocks comprise the bodies of procedures, entries, and modules, and may also be inserted anywhere a statement is allowed, except inside handlers. 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:

- Attempts to send or receive messages on a nil or dangling link.
- 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 replied.
- Hardware failures of various sorts.

Built-in exceptions are raised in the block in which they occur. If that block has no handler, the exception is raised in the next scope down 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 sending process.

User-defined exceptions are raised with the command

raise <exception name> ;

A user-defined exception must be declared in a scope visible to all the threads that use it. When raised, it will be felt by all and only those threads that have declared a handler for it in some scope on their current dynamic chain. 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.

Blocking Commands

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.

Discussion

Links

The notion of links is borrowed directly from Charlotte [1]. Charlotte in turn is a descendant of the Demos [6] and Arachne [7] operating systems. In Demos and Arachne a link is a capability to an input port [8]. In Charlotte, a link is an inseparable pair of capabilities, neither one of which may be copied. We have found links to be an invaluable abstraction, and would seriously consider their use in a programming language for other environments as well. We do not envision serious implementation problems with most of the distributed operating systems with which we are familiar.

Links allow a process to reason about its connections with the rest of the world on the basis of the services they provide, not the processes to which they are connected. For example, a client may hold a link to one of a community of servers. The servers 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 servicing its requests at a particular point in time. The client need not even be aware of such goings on.

We anticipate a great deal of link creation, movement, and destruction in Charlotte. In the example below, links are used to represent open files. We have used them to represent other physical devices as well, including blocks of memory for down-loading processes. We are aware of only one other language that provides a comparable degree of flexibility in process interconnections. NIL [9], in active use at IBM's T. J. Watson Research Center, performs all type checking at compile time. Our scheme has two major advantages:

(1) A process can possess large numbers 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.

Though run-time type checking will admittedly involve costs that compile time checking does not, we expect the amount of work involved to be insignificant in comparison to the overhead of communication. At the expense of absolute security we can use a hash function to reduce the self-descriptive portion of messages into a very small number of bits [10].

Synchronization Semantics

Liskov [11] describes three principal varieties of message synchronization:

No-Wait Send. A sender continues execution immediately, even as its message is beginning the journey to wherever it is going.

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

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 in 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 we allow a process to continue with other threads of control when a given one is blocked, and since we expect our machines to 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), inter-process 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.

We have chosen the remote invocation send over the 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. Since we expect most messages to need a reply, why not let the acknowledgment carry useful data?

There is some motivation for providing synchronization send as an additional mechanism. For messages that really need no reply, top-level acknowledgments can be sent by run-time support routines, rather than by the user's program, allowing us to unblock the sender after two fewer context switches on the receiving end. The savings are small, however, and we do not feel they justify cluttering the language with a second kind of send.

Explicit and Implicit Message Receipt

LYNX provides two very different means of receiving messages: the accept statement and the mechanism of bindings. We call the former explicit message receipt, and the latter implicit message receipt. In [12] we argue that both are essential in a practical programming language. Explicit receipt is most useful for the exchange of messages between active, cooperating peers, say a producer and a consumer. Implicit receipt more accurately reflects the externally-driven nature of most server processes. A language that provides only one option will have applications for which it is awkward, confusing, or both.

Some existing languages, notably StarMod [13], already provide both forms of message receipt. We go one step farther in LYNX by allowing a process to decide at run time which form(s) to use on which communication links.

The accept statement does not open a new scope in LYNX. We prefer not to have to copy message parameters into variables that remain visible after the scope of the accept is closed, the way one must in Ada [4].

Unlike most proposed languages with explicit receipt, we have not provided a mechanism for accepting a message on any one of a set of links. So far, we have found such non-determinism to be useful only in those cases where implicit receipt is the more appropriate approach. If at some point it proves necessary, we may add a non-deterministic variety of explicit receipt as well.

Multiple Threads of Control

There are several reasons for writing concurrent programs. Systems programs for a multicomputer must by their very nature be distributed. Applications programs may choose to run on multiple machines as well, in order to reduce execution time. Even on a single machine, however, many processes can most easily be written as a collection of largely independent threads of control. Many language designers have made this observation, and have allowed multiple threads of control to operate inside a single module and share that module's data. Generally, 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.

We feel this simulated parallelism is a mistake. 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 concurrently, we introduce race conditions that should not even exist; we force the programmer to provide explicit synchronization of even the most basic operations.

In Extended CLU [11] and StarMod [13], 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 Synchronizing Resources [14], 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.

We believe a much more attractive solution can be seen in the semantics of Brinch Hansen's Distributed Processes [15]. 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, or expensive message passing.

For the semantics of LYNX, we have adopted the Distributed Processes approach, with five extensions:

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

(2) A process may service external requests while waiting for one of its own.

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

(4) Blocked threads can be interrupted by exceptions.

(5) Modules and procedures may nest without restriction.

The fifth extension is, perhaps, the most controversial. As in Ada, it allows the sharing of non-global data. Techniques for managing the necessary tree of activation records are well understood [16]. We feel the added convenience of nested environments justifies the expense involved.

We anticipate some additional expense in the repeated evaluation of await-ed conditions. We expect our compiler to optimize the special case of waiting on a simple Boolean variable. The rules for context switching in LYNX allow such a variable to perform the work of a traditional Boolean semaphore.

Exceptions

We are aware of no precedent for our choice of semantics for user-defined exceptions. Given the existence of built-in exceptions, we found

the user-defined variety to be the simplest way to allow one thread to interrupt another. To demonstrate the use of exceptions, and to give a general impression of the flavor of LYNX, we end this section with an example.

The code below is meant to be a part of a file-server process. It 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, the file server binds that link to a number of procedures, one for each of the services it provides. One of those services, that of opening files, is shown in the example below.

Open files are represented by links. Within the server, each file link is managed by a separate thread of control. Context is maintained automatically from one request to the next. As suggested by Black [17], we adopt an asynchronous protocol in which bulk data transfers are always initiated by the producer (with connect) and accepted by the consumer. When a file is opened for reading, the file server plays the role of producer. We implement seek requests by raising an exception in the thread that is attempting to send data out over the link.

Clients close their files by destroying the corresponding links.

```
module fileserver (switchboard : link);
-- part of a no-frills file server
-- starts with single link to switchboard

entry open (filename : string;
    read, write, seek : Boolean) : link;
var
    filelnk : link;
    readptr, writeptr : integer;
    seeking : exception;

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

    entry write (data : bytes);
    begin
        put (data, filename, writeptr);
        inc (writeptr);
        reply;
    end write;

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

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

        if write then
            if seek then
                bind filelnk to writeseek;
            end;
            bind filelnk to write;
        end;

        if read then
            if seek then
                bind filelnk to readseek;
            end;
            loop
                begin
                    connect (
                        get (filename, readptr)
                        [] ) on filelnk;
                    inc (readptr);
                when seeking do
                    -- nothing; continue loop
                when filelnk.DANGLING do
                    exit;  -- leave loop
                end;
            end;  -- loop
        end;
    else  -- not available
        reply (nolink);  -- release client
    end;
    -- control will not leave open
    -- until nested entries have died
end open;

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

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

Conclusion

We have described a new distributed programming language specifically designed for the writing of systems programs for a multi-computer. Our language differs from previous proposals in three important ways:

(1) It introduces the notion of links. Links allow processes to examine and manipulate their interconnections at run time. Self-descriptive messages allow for full type security with no loss in flexibility.

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

(3) A control-flow mechanism similar to coroutines allows individual processes to be broken up into multiple threads of control. The predictability of context switches eliminates intra-module race conditions and simplifies the ordering of external requests.

Current Status

The design of LYNX is still in a state of flux. Among the issues we are continuing to consider are mechanisms for forwarding requests, for receiving asynchronous messages, and for performing non-type-checked input/output. The Charlotte kernel and servers are up and undergoing testing. Preliminary versions of the servers have been written in a local dialect of Modula [18], peppered with direct calls to the kernel communication primitives. An experimental LYNX compiler is under construction. The run-time scheduler and environment bookkeeping are operational. Language support for links has yet to be completed.

Acknowledgments

Marvin Solomon is a principal investigator for Charlotte (along with Raphael Finkel). Other researchers include Yeshiahu Artsy, Hung-Yang Chang, Aaron Gordon, Bill Kalsow, Vinod Kumar, Hari Madduri, Brian Rosenberg, and Cuiqing Yang. Many others support our efforts through their work on the larger Crystal project.

References

[1] 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).

[2] N. Wirth, "Modula: a Language for Modular Multiprogramming," Software--Practice and Experience 7 (1977), pp. 3-35.

[3] K. Jensen and N. Wirth, Pascal User Manual and Report, Lecture Notes in Computer Science #18, Springer-Verlag (1974).

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

[5] D. Beech, "A Structural View of PL/I," Computing Surveys 2:1 (March 1970), pp. 33-64.

[6] F. Baskett, 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.

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

[8] P. Cashin, "Inter-Process Communication," Technical Report 8005014, Bell-Northern Research (3 June 1980).

[9] 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 (27-29 June 1983), pp. 73-82.

[10] 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).

[11] B. Liskov, "Primitives for Distributed Computing," Proceedings of the Seventh ACM Symposium on Operating Systems Principles (December 1979), pp. 33-42.

[12] M. L. Scott, "Messages v. Remote Procedures is a False Dichotomy," ACM SIGPLAN Notices 18:5 (May 1983), pp. 57-62.

[13] R. P. Cook, "*Mod--A Language for Distributed Programming," IEEE Transactions on Software Engineering SE-6:6 (November 1980), pp. 563-571.

[14] G. R. Andrews, "Synchronizing Resources," ACM TOPLAS 3:4 (October 1981), pp. 405-430.

[15] P. Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept," CACM 21:11 (November 1978), pp. 934-941.

[16] D. G. Bobrow and B. Raphael, "New Programming Languages for Artificial Intelligence Research," Computing Surveys 6:3 (September 1974), pp. 153-174.

[17] A. P. Black, "An Asymmetric Stream Communication System," Proceedings of the Ninth ACM Symposium on Operating Systems Principles (10-13 October 1983), pp. 4-10.

[18] 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).