Comput. Lang. Vol. 16, No. 3/4, pp. 209–233, 1991
Printed in Great Britain.

0096-0551/91 $3.00 + 0.00

THE LYNX DISTRIBUTED PROGRAMMING LANGUAGE: MOTIVATION, DESIGN AND EXPERIENCE

MICHAEL L. SCOTT
Department of Computer Science, University of Rochester, Rochester, NY 14627, U.S.A.

(Received 8 March 1990; revision received 4 September 1990)

Abstract—A programming language can provide much better support for interprocess communication than a library package can. Most message-passing languages limit this support to communication between the pieces of a single program, but this need not be the case. Lynx facilitates convenient, typesafe message passing not only within applications, but also between applications and among distributed collections of servers. Specifically, it addresses issues of compiler statelessness, late binding, and protection that allow run-time interaction between processes that were developed independently and that do not trust each other. Implementation experience with Lynx has yielded important insights into the relationship between distributed operating systems and language run-time support packages and into the inherent costs of high-level message-passing semantics.

Distributed programming languages Message passing Remote procedure call Late binding Server processes Links

1. INTRODUCTION

A programming language has clear advantages over a library package for communication between processes in a distributed environment.† These advantages include the possibility of special syntax (more than just a set of subroutine calls), direct use of program variables and types in communication statements, message type checking, exception handling, and support for concurrent conversations. Unfortunately, most existing distributed languages are better suited to communication between the processes of a single application than they are to communication between processes that are developed independently. Such independent development is characteristic both of the systems software for multicomputers and of the applications software for geographically-distributed networks. Lynx [1, 2] is a message-passing language designed to support both application and system software in a single conceptual framework. It extends the advantages of language-based communication to processes that are designed in isolation, and compiled and placed in operation without knowledge of their peers. This paper provides an overview of Lynx, from the problems that led to its creation through the experience resulting from its implementation and use.

Motivation for Lynx is discussed in Section 2. Lynx was developed at the University of Wisconsin, where it was first implemented on the Charlotte multicomputer operating system [3, 4]. Charlotte was designed without Lynx, but experience with a conventional library interface to the kernel suggested that language support for communication could make the programmer's life much easier. Particularly troublesome was the construction of operating system server processes, which needed to communicate conveniently, safely, and efficiently with an ever-changing pool of clients, most of whom could be expected to have been written long after the server was placed in operation. Lynx was designed to provide a high degree of support for dynamic changes in interconnection topology, protection from untrusted processes, and concurrent, interleaved conversations with an arbitrary number of communication partners. To facilitate the construction of servers, Lynx was designed to provide these benefits without depending on any sort of global information at compile time.

The Lynx compiler is a pure translator: it requires no input other than a source program and produces no output other than an object program. In particular, it does not depend on a database

†Throughout this paper, the term "process" is used to denote a heavyweight entity with its own address space, supported by the operating system. Active entities in the same address space will be called lightweight threads of control.

of type definitions or interface descriptions in order to enforce type consistency for messages. A novel application of hashing [5] provides efficient run-time type checking at negligible cost and at a very high level of confidence.

A description of language features appears in Section 3. This is followed in Section 4 by a pair of example applications: a file system server and a game-playing program. Section 5 provides a more detailed rationale for the language design, and relates it to previous work. Within the context of independent compilation, Lynx supports topology changes and protection with a virtual-circuit abstraction called the *link*. It maintains context for multiple conversations by combining message passing with the scheduling of lightweight threads of control. A link is a symmetric two-directional channel, like a pair of tin cans connected by string. The cans themselves (link ends) are first-class objects that can be created, destroyed, stored in data structures, and passed in messages. It is by passing them in messages (dragging the strings behind them) that the process connection graph is changed. Threads are a program structuring tool that allows a sequential execution path to be associated with each logically separate conversation. A file server, for example, might have a separate thread of control for each of its open files. Each thread could then use straight-line code to perform operations on behalf of a client. Special operations (e.g. seeks), could be performed by nested threads that share file-specific data structures. Lynx allows these threads to be created automatically, in response to incoming requests.

At the University of Rochester, Lynx has been ported to the BBN Butterfly multiprocessor and its Chrysalis operating system. Designs or partial implementations have been developed for four additional systems. Experience with these implementations has led to important insights into the relationship between a language run-time package and the underlying operating system [6]. A detailed performance study of the Chrysalis implementation has helped to provide a deeper understanding of the inherent costs of message-passing systems [7]. These lessons are summarized in Section 6.

2. MOTIVATION

Motivation for Lynx grew out of experience with the Charlotte distributed operating system [3, 4]. Like many other systems developed in the 1970s and early 1980s, Charlotte was designed to provide many of its services in user-level processes, rather than in the kernel.† The first generation of Charlotte servers was written in a conventional sequential language (Modula-1 [8], sequential features only), augmented with procedure calls to access kernel message-passing primitives. Problems with this approach soon became apparent, and suggested the need for a special programming language. Section 5 discusses these problems in detail, considering candidate solutions and justifying the Lynx approach. The remainder of this section enumerates the major issues, to establish context for the upcoming language description.

Problems encountered in the construction of Charlotte servers were of three main types, all of which stem from invoking communication primitives through a subroutine library interface:

Type checking and the marshalling of message parameters

Charlotte kernel calls specify messages in the form of a location and a length. Buffer management is the programmer's problem, and the data sent in messages must be gathered and scattered explicitly, generally by using casts to overlay a record structure on an array of bytes. The code is awkward at best and depends for correctness on programming conventions that are not enforced by a compiler.

Error handling

Every Charlotte kernel call returns a status value that indicates whether the requested operation succeeded or failed. Different sorts of failures result in different values. A well written program must inspect every status and be prepared to deal appropriately with every possible value. It is not unusual for 30% of a carefully written server to be devoted to error checking and handling. Even an ordinary client process must handle errors explicitly, if only to terminate when a problem occurs.

†Charlotte servers include a process and memory manager (the *starter*), a server-like front-end for the kernel (the *kernjob*), a command interpreter, a process connection facility, two kinds of file servers, a name server (the *switchboard*), and a terminal driver.

Conversation management

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—it must be able to attend to other conversations. Charlotte servers therefore adopt an alternative program structure in which a single global loop surrounds a case statement that handles arbitrary incoming messages. This explicit interleaving of separate conversations is very hard to read and understand.

Users of distributed systems other than Charlotte have encountered similar problems. As a result, most of the message-based operating systems still in use employ *stub generators* that augment the kernel call interface with a customized "wrapper" routine for each type of message in a program. Birrell and Nelson's Lupine stub generator [9] and the Accent Matchmaker [10] are particularly worthy of note. Stubs can eliminate many of the problems with straightforward use of a kernel call library, but they still limit communication to a procedural interface, and they address the second and third problems above only in the context of a host language with well-designed mechanisms for exception handling and internal concurrency, something not available in the context of the Charlotte project.

Upon surveying the state of the art in distributed programming languages [11], it became apparent that most existing notations had been oriented toward communication among the processes of a single distributed program, and that new problems would arise when attempting to communicate across program boundaries—with servers, for example. A language for inter-program communication needs to

(1) perform type checking on messages, even when the identities of communication partners are not known at compile time;
(2) support dynamic configuration of the connections between processes, even when some of the parties participating in the configuration are not aware of the use to which a connection will be put;
(3) protect processes from each other, allowing them to choose their communication partners and the extent to which they trust them; and
(4) provide a lightweight thread mechanism as a conversation structuring tool (as opposed to a means of expressing genuine parallelism), with a careful integration of the primitives for thread management and interprocess communication.

These issues guided the development of Lynx, and are woven throughout the material contained in the following section.

3. LANGUAGE DESIGN

Links give Lynx its name, and constitute the channels over which messages travel between processes. Implicit receipt of messages provides the principal mechanism for creating threads of control within a process. These two concepts are discussed in the first two subsections below, and are followed by a more detailed description of the mechanisms for dynamic configuration of the connections between processes and for checking message types. The integration of communication and thread management facilities forms the subject of the last two subsections, which address scheduling and referencing environments.

3.1. Links

Processes in Lynx are assumed to be independent and autonomous. Each process is separately compiled. At run time, processes communicate only by sending messages to each other over two-directional communication channels called *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 word "entry" introduces a template for a remote operation. The general syntax is

entry opname (*request_parameters*):*reply_parameter_types*;

In this case, the transfer entry has no reply parameters.

An entry header can be followed by a body of code, or by the word "remote." In our case, we have used the latter option because the code for transfer is in another process. Like the word "forward" in Pascal, "remote" can also indicate that the code will appear later in the current process, either as a repeated entry declaration or as the body of an *accept* statement, as in the consumer below.†

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 as a whole 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 provide an operation requested by the process at the other end of a link. In our example, the producer uses a connect statement to request a transfer

†This overloading of the word "remote" has proven confusing to users, and is probably a mistake.

operation over its link to the consumer, and the consumer uses an accept statement to provide this operation.

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

The reply clause at the end of the accept 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. The accept and reply portions of the statement are syntactically linked. Arbitrary statements can be nested inside, including additional accept-reply pairs. In our example, the consumer has nothing it needs to do before replying.

A link can be thought of as a *resource*. In our example neither the consumer nor the producer can name the other directly. Each refers only to the link that connects them. The consumer, having received all the data it wants, might pass its end of the link on to another process. Future transfer operations would be provided by the new consumer. The producer would never know that anything had happened.

A variable of type link really identifies a link *end*. Link ends are created in pairs, by a built-in routine called newlink. Our producer/consumer pair could be created with the following sequence of statements:

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

To make it easy to write sequences of code such as this one, newlink returns one of the link ends as its function value (here passed on immediately to the consumer) and the other through a reference parameter (here saved temporarily in L so that it can be passed to the producer in the second call to startprocess).

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.

```
startprocess ("consumer", newlink (L));
startprocess ("buffer", L, newlink (M));
startprocess ("producer", M);
```

Code for a buffer process appears in Section 3.5.

3.2. Implicit receipt

In our producer/consumer example, each process contains a single thread of control. In the consumer, this thread accepts its transfer requests explicitly. It is also possible to accept requests *implicitly*, and the choice between the two approaches depends largely on whether we view the consumer as an active or a passive entity.

If we think of the producer and consumer as active peers, then it makes sense for the consumer to contain a thread that "deliberately" waits for data from the producer. If we choose, however, to think of the consumer as a server (a spooler for a printer, perhaps), then we will most likely want to write a more passive version of the code—one that is driven from outside by the availability of data. Since a demand-driven spooler is likely to have multiple clients, it also makes sense to give

each incoming request to a separate thread of control, and to create those threads automatically. Our consumer can be re-written to use implicit receipt as follows:

```
process consumer (producer:link);
type data = whatever;
procedure consume (info:data);
begin
  —whatever
end consume;
entry transfer (info:data);
begin
  reply; —allows producer to continue
  consume (info);
end transfer;
begin—consumer
  bind producer to transfer;
end consumer.
```

Here we have provided a begin...end block for the transfer entry procedure, instead of declaring it remote. Each connect to transfer will create a new thread of control in this version of the consumer. As with the reply portion of an accept statement, the reply statement of the entry causes the run-time support package to unblock the thread of control (in the producer) that requested the current operation. The replying thread continues to exist until it runs off the end of its entry. The run-time system is required to detect a thread that attempts to reply twice, or erroneously terminates before replying. The producer shown above can be used with either version of the consumer, without modification.

The bind statement serves to create an association between links and entry procedures:

bind *link_list* to *entry_list*;

Only those operations provided by accept statements and bindings to entries can be requested by the process at the far end of a link. Connect statements that request a non-existent operation will cause an Ada-like exception in the requesting thread of control.

Bindings can be broken as well as made:

unbind *link_list* from *entry_list*;

The ability to manipulate bindings at run time is a powerful mechanism for access control. Each process has complete control over which of its communication partners can invoke which operations at which points in time. Reference [1] contains a Lynx solution to the classic readers/writers problem. This solution permits a client to obtain read and/or write access to a resource and perform an arbitrary sequence of operations before relinquishing that access. The sequence of operations need not be known at the time that access is obtained; a client can, for example, obtain read access, read an index, and read a location calculated from that index in one protected session. Solving the same problem in Ada [12] requires a complicated system of unforgable *keys*, implemented in user code.

It is the ability of a server to refer to links by name that permits it to implement access control. A server can, if desired, 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. Of course, a server has no way of knowing which *process* is attached to the far end of a link, and it has no way of knowing when that far end moves, but this is in keeping with the concept of process autonomy. A link to a client represents an abstraction (a connection over which to provide a service) every bit as much as a link to a server represents a connection over which a service is provided. In fact, it is entirely possible for two processes to act as servers for each other, with a single link between them. A file server, for example, might use a link to a sort utility in order to maintain indices. The sort utility for its part might use the file server as a place to store large data sets.

Symmetric, two-directional 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 has only a localized effect: it kills the current thread. In a server, this serves to terminate the conversation with the client whose communication failed.

3.3. Link movement

To move a link end in Lynx, a process need only enclose it in a message (via connect, accept, reply, or startprocess). Once the message is received, the sending process can no longer use the transferred link, but the receiving process can. The compiler provides the run-time system with enough information about types that this transfer is guaranteed to work for messages containing arbitrary data structures (including variant records) that might have links inside.

There are many reasons to change the connections between processes at run time. A link between a server and a client can be passed on to a new client when the first one does not 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. In a large distributed environment, many servers are likely to be implemented by dynamic *squads* of processes [13]. These processes may move their end of a client link frequently, in order to connect the client to the member of their group best able to serve its requests at a particular point in time.

Certainly a newly-created process must be connected to existing processes that can provide it with input and output. In a single, large application, it is also common for computation to move through a series of distinct phases, each of which requires a different set of processes and connections. In a robust, geographically distributed system, a process that is unable to obtain a service from its usual source (because of hardware failures, for example, or overloaded communication lines) may wish to connect to an alternative server.

One of the most common uses of link movement is shown in Fig. 1. A *name server* process, or *switchboard*, maintains a registry 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. The command interpreter, or shell, is likely to provide each newly-created process with a link to the switchboard, from which it can obtain links to whatever other servers it may need.

To find, for example, a line-printer spooler, client C would send a message to the switchboard:

```
connect find_server ("lp_spooler"|spooler_link) on switchboard_link;
```

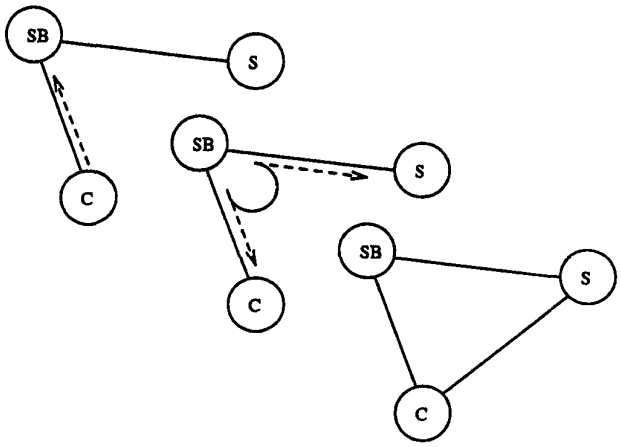


Fig. 1. Querying the switchboard.

Upon receiving this request, the switchboard would scan its registry for a server that has advertised the name "lp_spooler". Assuming such a server exists, the switchboard would create a new link, pass one end on to the server in a newclient message, and return the other end to the client:

```
entry find_server (server_name:string):link;
var server, rtn:link;
begin
  server:= lookup (server_name);
  if server=nolink then reply (nolink);
  else connect newclient (newlink (rtn)|) on server; reply (rtn);
  end;
end find_server;
```

Newlink returns one link end as its function return value (which here becomes a parameter to the newclient operation) and the other through a reference parameter. Each server that wishes to accept new clients must provide the switchboard with a link over which it is willing to accept newclient requests.

3.4. Type checking

To a large extent, links are an exercise in delayed decision making. 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.

Run-time type checking of messages provides three distinct advantages over compile-time checking of connections:

(1) A process can hold a large number of links without knowing what 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. It need not declare a conservative superset of those types at compile time, nor ever worry about receiving a message of a currently-inappropriate type at run time.
(3) With an appropriate choice of semantics for type equivalence, the compiler can be designed to work without a global database of types. Processes can be compiled in any order, at any site in a distributed environment, without requiring the compiler to maintain state between compile runs, or to keep that state consistent.

Type checking in Lynx is based on *structural* equivalence. Two types are considered the same if they contain the same internal structure—the same set of primitive types composed with the same higher-level type constructors. The compiler can provide the run-time system with a "canonical" representation of each type, so that type checking becomes a simple comparison for equality of canonical forms.

Since canonical forms can be of arbitrary length, run-time comparisons are potentially costly. To minimize this cost, the Lynx compiler uses a hash function to compress its type descriptions into 32-bit codes [5]. Hashing reduces the cost of type checking to half a dozen instructions per remote operation. It introduces the possibility of undetected type clashes,† but with a good hash function the probability of this problem is less than one in a billion.

A second potential 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 run time in Lynx. This cost, too, is small, and easily justified by the type system's simplicity and flexibility. As a practical

†In the event of an undetected type clash, the sender and receiver of a message will apply incompatible interpretations to the data in a message, with unpredictable results.

matter, we tend to rely on shared declaration files to ensure that run-time clashes are rare. We catch most type errors at compile time and the rest (with high probability) at run time, much more easily than we could catch all of them at compile time.

A final cost of the Lynx approach to types is the somewhat liberal 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 unrelated. Lynx shares this form of checking with many other languages, including Algol-68, Smalltalk, Emerald, and many dialects of Pascal. We are happy with structural equivalence. No type system, no matter how exacting, can *ensure* that messages are meaningful. Type checking can be expected to reduce the likelihood of data misinterpretation, not to eliminate it.

3.5. Thread management

Though the implicit-receipt version of our consumer process 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 (consumer, producer: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; —not full
  buf[firstfree] := info;
  firstfree := firstfree % size + 1;
  reply;
end transfer;
var info:data;
begin
  firstfree := 1;
  lastfree := size;
  bind producer to transfer;
  loop
    await lastfree % size + 1 <> firstfree; —not empty
    lastfree := lastfree % size + 1;
    info := buf[lastfree];
    connect transfer (info|) on consumer;
  end;
end buffer.
```

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 "calling" an entry locally.

The threads of control within a single process do not execute in parallel; each process continues to execute a single thread until it *blocks*. The process then takes up some other thread where it last left off. If no thread is runnable, then the process waits for completed communication to change that situation.

Threads may block (1) for communication (connect, accept, reply), (2) for completion of nested threads (when leaving a shared scope), (3) for a reply from a locally-created thread, and (4) for an explicitly await-ed condition. In the bounded buffer example, the await statement blocks the current thread until the buffer is non-empty or non-full, as appropriate. There is no need to worry about simultaneous access to buf, firstfree, or lastfree, because the coroutine-style semantics

guarantee that only one thread can execute at a time. In the file server example of Section 4.1, the coroutine semantics also guarantee that threads are at predictable places in the code when an exceptional condition occurs.

Of course, 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 may not be immediately obvious where those blocking statements are, but the compiler can help by identifying them in listings. 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 binary semaphore.

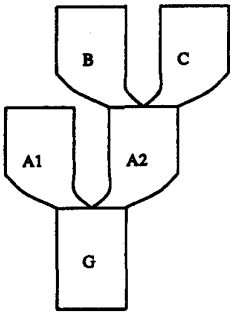
A link end may be bound to more than one entry. The bindings need not be created at the same time, nor are they incompatible with use of the link end in one or more outstanding accept statements. It is therefore possible for separate threads to carry on independent conversations on the same link concurrently. The run-time support package keeps track of the correspondence between requests and replies, routing each to the appropriate thread. As an example, suppose the run-time system implements the startprocess statement by sending a request to a process-creation server that is itself written in Lynx. Each such request might create a new thread of control within that server. The server would need a link to a file server, over which to read executable files; concurrent process-creation requests, managed by different threads, would share that link 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 requests, by contrast, can be unexpected or unwanted. 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 exception is raised in the thread that executed the connect statement at the other end of the link.

3.6. Stack management

The syntax of Lynx allows entries to be declared at any level of lexical nesting. Non-global data may therefore be shared by more than one thread of control. The file server example in the following section uses one thread of control for every open file. Additional, nested threads implement writeseek, stream, and readseek operations. This sharing of non-global data can be conceptualized as a so-called *cactus stack*.

A cactus stack is actually a *collection* of stacks, prefixes of which may be shared. Figure 2 illustrates a situation in which two threads have been created in entry A, and one of them has nested threads in entries B and C. Both of the threads executing A will have access to global variables. They will, of course, have access to their own local variables as well. If entry A serves to implement




```
process G ...
    entry A ...
        entry B ...
        entry C ...
    begin -- A
        ...
        bind clientlink
            to B, C;
```

Fig. 2. A cactus stack.

```
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;
        if writeflag then
            if seekflag then bind filelnk to writeseek; end;
            bind filelnk to stream;
        end;
        if readflag then
            if seekflag then bind filelnk to readseek; end;
            loop
                begin
                    connect stream (get (filename, readptr) | ) on filelnk;
                    readptr +:= 1;
                when seeking do
                    -- nothing; try again at new location
                when REMOTE_DESTROYED do
                    exit;  -- leave loop
                end;
            end;  -- loop
        end;  -- if readflag
    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 newclient, open; reply;
end newclient;
begin  -- main
    bind switchboard to newclient;
end fileserver.
```

Fig. 3. Stream-based file server in Lynx.

a non-trivial service, an instance of A (call it A2) may provide its client with a call-back facility by binding a link to nested entries. Incoming requests for those entries will create concurrent threads inside the context of A2. These threads will share access to A2's local variables.

In the illustration, each segment of the cactus stack represents a subroutine or entry activation. Each branching point corresponds to the creation of a nested thread of control. There is, therefore, one thread per leaf of the structure, and one thread for every internal segment (other than the root) that lies at the top of a trunk. From the point of view of any one thread, the path back down to the root looks like a normal stack.

A Lynx thread blocks if it reaches the end of a scope in which nested threads or bindings are still active. This rule ensures that segments never "disappear" out of the middle of the structure. In Fig. 2, a thread executing entry A will be suspended automatically when it reaches the end of its scope, continuing only when there is no further possibility of creating threads in entries B and C (e.g. when clientlink is destroyed).

4. EXAMPLES

4.1. A simple stream-based file server

A realistic example of the use of threads and links can be seen in Fig. 3, which contains simplified code for a file server process under Charlotte. The original Charlotte file server was written in a sequential subset of Modula, with library calls for communication. It comprised just under 1000 lines of code, and was written and rewritten several times over the course of a 2-year period. It was a constant source of trouble. By comparison, the Lynx fileserver is just over 300 lines, and was written in only 2 weeks. It would have required even less time if it had not been undertaken concurrently with debugging of the language implementation.

Each of the problems described in Section 2 appeared in the original server. Buffers were sometimes modified while the kernel was still busy sending them, or read before the kernel had finished receiving them. Changes to the server interface introduced undetected incompatibilities with existing clients, leading to mysterious behavior due to incorrect interpretation of messages. Equally mysterious behavior resulted when kernel call failures went unnoticed due to lack of return code checking. Changes that required the fileserver to communicate with another server were very hard to make: the easy alternative was to block the entire fileserver while the request completed; the better approach often required major reorganization of the code to get back into the central message dispatch loop in the middle of a deeply nested function. The Lynx fileserver, in contrast, is easy to read and maintain.

The code of Fig. 3 employs the newclient convention of Section 3.3. We have written the server to take a single initial argument: a link to the switchboard name server. Additional clients are introduced by invocations of newclient over links from the switchboard or from clients. When a newclient request is received (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. We have adopted the convention that data transfers are initiated by the producer (with connect) and accept-ed 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.

In addition to a conventional mechanism for raising exceptions in a single thread of control, Lynx also permits one thread to cause an exception in another. In the file server example, this facility is used to handle seek requests in a file that is open for reading. Under the normal stream protocol, the file server will always attempt to transfer a block (with "connect stream...") as soon as the previous block has been received. In order to read blocks out of order, the client invokes a readseek operation. The thread that provides this operation uses an announce statement (line 20) to interrupt the thread (line 36) that is trying to send the wrong block.

Announce causes its exception to be felt in all and only those threads that have a matching handler on their current call chain. Because the seeking exception is declared inside the scope of

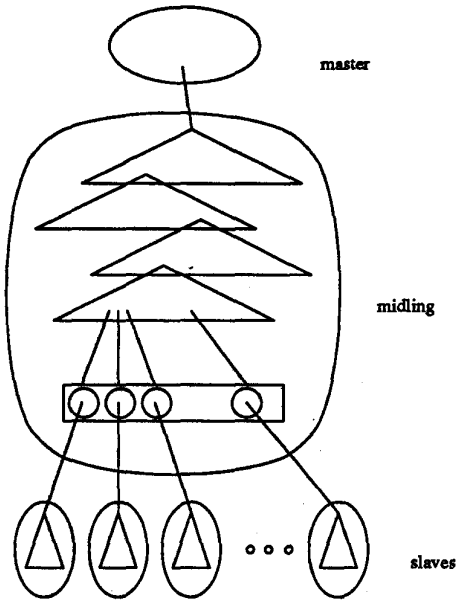


Fig. 4. Checkers program structure.

the open entry, there is a separate such exception for every open file. When the exception is announced, the sending thread retries its connect, using the updated file pointer. Since incoming requests (and readseek requests in particular) are received only when all threads are blocked, the thread that provides the readseek operation can be sure that the thread that is streaming data must be stopped at its connect statement. No race conditions can occur.

To close an open file, a client need only destroy the link that represents the file.† Like an incoming message, destruction of a link by the process at the other end is noticed when all threads are blocked. Any thread currently attempting to use the 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.

4.2. A distributed game-playing program

Using an implementation of Lynx on the BBN Butterfly multiprocessor, we have created a distributed program that plays the game of checkers (draughts). Since our principal goal was to evaluate Lynx and not to investigate the design of parallel algorithms, we adopted an existing parallelization of alpha–beta search [14].

The basic idea behind the algorithm can be seen in Fig. 4. There are three different kinds of processes. One process (the "master") manages the user interface (in our case, this is a graphic display under the X window system). A second process (the "midling") manages the parallel evaluation of possible moves. A third kind of process (the "slave") performs work on behalf of the midling. There is only one master and one midling. Performance is maximized when there is one slave for every available processor.

Within the midling, one thread of control explores the first few level of the game tree. It constructs a data structure describing all of its possible moves, all of the possible subsequent moves by its opponent, all of its possible moves after that, and so forth. At a given depth in the tree (typically four or five levels), it enters board positions into a queue of work to be performed by slaves.

†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. Future attempts to use them may result in an exception or, depending on the implementation, use of an unintended link.

Each slave is represented by a separate thread in the midling. That thread repeatedly removes an entry from the work queue, sends it to a slave, and waits for the result. When that result comes back it updates the game tree, performs any necessary pruning (to throw away moves that are now known to be sub-optimal), and obtains a new entry from the queue. In order to avoid storing all of the top few levels of the (very large) game tree at once, the thread that creates the data structure blocks when the work queue is full. Game tree nodes are thus created on demand. Likewise, the threads that dispatch work to slaves will block when the work queue is empty. Despite the fact that the checkers player is a self-contained program, the midling bears a strong resemblance to a server. It would have been significantly more difficult to write the midling with a single thread of control.

One consequence of the communication semantics of Lynx is that a process does not notice incoming messages until all of its threads are blocked. There is no way to receive a message asynchronously or to allow a high-priority message to interrupt the execution of lower-priority "background" computation. In the checkers program, performance is likely to improve if a slave can be interrupted when the midling discovers that its subtree has been pruned, or perhaps when it discovers new information that will help the slave do more pruning internally. For cases such as this, Lynx provides a low-cost polling function that can be used to determine if messages are pending. Slaves execute the statement

```
await idle;
```

at the top of an outer loop. Update messages from the midling are therefore received within a reasonable amount of time.

Informal experiments with various problem parameters (number of tree levels in the midling, size of subtrees evaluated by slaves, frequency of update messages, etc.) have produced 10–20-fold speedups with 100 working slaves. The primary limiting factor appears to be that many of the subtrees that are evaluated in parallel would have been pruned off and never explored by the standard sequential algorithm.

5. DISCUSSION

As described in Section 2, motivation for Lynx grew out of experience with Charlotte, and in particular with three types of problems that arose in that experience. Section 5.1 attributes these problems directly to the use of subroutine libraries for interprocess communication. It notes that similar problems do not arise (at least not to such an extent) in procedural interfaces to more traditional kernel services, and explains how communication mechanisms can be improved by programming language support. Section 5.2 then argues that existing distributed languages are best suited to communication among the processes of a single parallel program, and discusses the techniques used in Lynx to extend the advantages of language-supported communication into a broader context.

5.1. Message-passing languages

Subroutines provide the natural mechanism for trapping into an operating system kernel for service, and for most services this mechanism works well. Experience suggests, however [4, 9, 10, 15], that users do not find it acceptable for interprocess communication. The crux of the problem is that communication is significantly more complicated, from the user's point of view, than are most other kernel services. Recall the problems cited in Section 2:

Type checking and the marshalling of message parameters

Like file system read and write operations, library-based communication primitives generally transfer uninterpreted streams of bytes. The desire to impose structure on those bytes has often been a motivation for language-level file system interfaces, and that motivation applies even more strongly to messages. Communication statements in a language can make direct use of program

variables, eliminating the need to think about message buffers. The compiler for a message-passing language can generate code to *gather* and *scatter* parameters. It can enforce type consistency with mechanisms used for separate compilation.

Error handling

Interprocess communication is more error-prone than other kernel operations. Message operations may fail due to hard errors or to program bugs in a communication partner (failure to set appropriate permissions, for example, or to receive within an expected time frame). Fault-tolerant algorithms may allow a process to recover from many kinds of detectable failures, but it can be awkward to deal with those errors in line by examining kernel call return codes. A programming language can provide facilities for exception handling outside the normal flow of control. A single exception handler can protect a large number of communication statements. Certain kinds of errors may even be handled in the run-time support package, without ever becoming visible to the programmer (a process may have moved, for example, and the run-time package can obtain a forwarding address).

Conversation management

While a conventional sequential program rarely has anything interesting to do while waiting for a kernel call to complete, a process in a distributed environment is much more likely to be budgeting its time among multiple activities. A server, for example, may be working on behalf of multiple clients at once. It cannot afford to be blocked while waiting for a particular client to respond. The kernel can help by providing non-blocking sends and receives, but then the server begins to resemble a state machine more than it resembles straight-line code. The inevitable interleaving of separate conversations leads to very obscure programs. With appropriate facilities for concurrency, a programming language can allow separate conversations to be handled by separate threads of control. The run-time support package can be designed to include a *dispatcher* routine that examines each incoming message and makes it available to the appropriate thread automatically.

In a more general sense, a message-based programming language can ease the life of the programmer by providing special communication syntax or by implementing useful side effects for communication statements. It would be difficult, for example, to provide the functionality of an Ada *select* statement [16] without its distinctive syntax. The *select* system call of Berkeley Unix, for example, is much less convenient to use. A less widely appreciated feature of Ada is its carefully-designed semantics for data sharing between tasks. The language reference manual requires a "shared" variable to have a single, consistent value only at the times when tasks exchange messages. An Ada implementation can choose to replicate variables at multiple sites and can allow the copies to acquire inconsistent values, so long as it reconciles the differences before the programmer can detect them—i.e. before the variables can be compared to a value in a message. In a similar vein, the compiler for NIL [17] tracks the status of every program variable and treats a variable that has just been sent in a message as if it were uninitialized. This facility allows the run-time system to implement message passing on a shared-memory machine by moving pointers, without worrying that a sender will subsequently modify the variables it has "sent". In Argus [18], one can send messages between processes that use completely different implementations of a common abstract type. The compiler inserts code in the sender to translate data into a universal, machine-independent format. It inserts code in the receiver to translate that data back into an appropriate local version of the abstraction. Argus also arranges for each remote operation to be executed as a nested atomic transaction.

The advantages of language-level communication can be realized either by desiging a language from scratch or by augmenting an existing language. Stub generators typify the latter approach. Stubs can diminish or eliminate several of the problems noted above, but they lack the full power of a language-based approach. Non-procedural syntax and special side effects are not available. Type checking requires an external mechanism (outside the compiler and the stub generator) to ensure that communication partners are compiled with respect to the same typed interface description. Errors of interest to the programmer can be propagated out of stub routines cleanly only with the help of a good exception-handling mechanism, which many host languages lack. High-quality conversation management requires both a lightweight thread facility (either in the

host language or as part of a library package) and an appropriate synchronization mechanism.† Each of these problems is easy to address when designing a language from scratch.

5.2. Inter-Program Communication

The bulk of the literature on distributed languages focuses on the needs of distributed programs—collections of processes that are designed to work together and that constitute the pieces of a single, coherent whole. There are equally important scenarios, however, in which communication must occur between processes that were developed independently. Distributed systems software provides one class of examples. Large-scale distributed applications provide another.

In a distributed operating system the communication activity of a server process is at least as complicated, and often more complicated, than that of any user application. Furthermore, the presence of servers means that even the most self-contained of programs is likely to need to communicate occasionally with an independently-developed process about which it knows very little. Language support can make this communication much more convenient and safe, particularly if it matches the style of communication used *within* the program. In fact, the more one moves away from the centralized model of a traditional operating system and toward a distributed collection of servers, the harder it becomes to draw the line between one program and the next.

Consider a large-scale application that spans a geographically distributed collection of machines. An airline reservation system is an obvious example. It is possible to conceive of such an application as a single distributed program, but the concept wears thin when one considers the large number of institutions involved. It is one thing to talk about a program developed by a single organization (the airline, say) and running on machines owned and operated by that organization. It is quite another to talk about a program that has pieces under the control of a thousand different travel agencies, possibly written in different languages and running on different types of hardware. If we consider the subject of automatic teller machines and the electronic transfer of funds between competing, autonomous banks, the concept of a single distributed program breaks down altogether. As internets proliferate, the software required for network management and routing is also developing into a multi-program distributed application. Further examples can be found in the Defense Department's need for global information gathering and communication, and in similar applications in weather forecasting and ground control for spacecraft.

When compared to a multi-process program, the pieces of a multi-program application have unusually stringent needs for compile- and run-time flexibility, protection, and conversation management.

Flexibility

As noted in Section 3.4, run-time checking of message types allows a process to hold and manipulate links even when it does not know what types of messages they may carry. It also allows a link to be used for more than one type of message without compromising the quality of type checks. The compile-time alternative would lead to coarser checking by associating a type with each link, rather than each message.

Regardless of whether type checking occurs at compile time or run time, server processes must be able to communicate with clients that did not exist when the server was designed. The obvious way to specify types for this communication is to create for each server an interface description that can be used when constructing clients. Unfortunately, maintaining a consistent set of interface descriptions across distributed sites becomes a non-trivial database management problem as soon as there is a significant number of servers or sites. If the compiler insists that a client be compiled with exactly the same set of declarations that were used to compile the server (this amounts to insisting on *name equivalence* for types), then the database problem becomes particularly severe: it requires unforgable, globally consistent version numbers that change when the data changes but not when it is copied.

---

†Not all concurrent languages provide good facilities for writing a dispatcher. If the variety of messages for which a thread might be waiting is not fixed at compile time (if, for example, threads are able to wait for a message from a specific sending process, or with contents that satisfy some predicate), then the number of synchronization conditions on which a thread can wait must also be unlimited, or (equivalently) it must be possible to maintain a table of threads and desired message types and then unblock specified threads by name. A static set of synchronization conditions (as, for example, in Modula-1 [8]) does not suffice.

Upward compatibility is also a problem. If a new comment is added to the interface description for the file server, we will certainly want to avoid recompiling every process that uses files. If a new routine is added without changing the behavior of the rest of the interface, we should also avoid recompilation. If changes are made to certain routines but not others, we should recompile only those processes whose behavior would otherwise be incorrect. It is possible to build a compiler that incorporates a formal notion of upward compatibility [19]. Such a compiler could implement run-time checking of name equivalence for types, but its construction would not be easy (even in the absence of multiple sites), and its checking would likely be costly. Lynx side-steps these problems by using *structural* type equivalence: no central database of types is needed, upward-compatible programs run without recompilation or sophisticated version tracking, and hashing makes run-time checks cheap.

To facilitate run-time changes in interconnection topology, Lynx makes the link an explicit first-class object, and provides variables that name a link end. A link end can be used to represent an abstract *resource* that is distinct from both the process(es) that implement it and the operations it provides. Examples of resources include a file, a bibliographic database, a print spooler, and a process creation facility.

Most other languages that allow connections to be reconfigured use variables that name processes or remote operations. The problem with naming processes is that a resource may be provided by a collection of processes; a distributed server may prefer that a user communicate with different constituent processes at different times, in order to balance workload or minimize communication costs. If users address their messages to processes, then the server cannot effect topology changes without informing the user; this violates abstraction. The problem with naming remote operations is that a resource may provide different sets of operations to different clients, or at different points in time. Even with facilities for bundling related operations (such as the resource and operation capabilities of SR [20]), the set of operations provided by an abstraction must be known to every client; servers cannot change the set of available operations to reflect changes in the state of the abstraction or to implement access control. If the resource is passed among clients, the desire for information hiding suggests that each client should be aware of only the operations it needs.

Protection

Pieces of a multi-program application cannot afford to trust each other. Even if malice is not an issue (as a result, let us say, of external administrative measures), each process must be able to recover from arbitrary errors on the part of its communication partners. Lynx provides such protection by incorporating message passing into a general-purpose exception-handling mechanism, with a built-in exception for each type of system-detectable error. It will always be possible, of course, for a process to send messages with incorrect data, but no language could prevent it from doing so. Errors propagated to the user in Lynx include type clashes, requests for an unavailable operation, termination of a communication partner, or destruction of a link that is currently in use (this latter error subsumes both hardware failures and software-requested destruction).

The concept of an unavailable operation arises from run-time type checking and from the ability of a server to control the set of operations that are available over each individual link at every point in time. As noted in Section 3.2, the ability to provide different operations over different links allows a server to differentiate between clients—to provide them with differing levels of service or extend to them differing levels of trust. In addition, the ability to change the set of operations available over a given link at different points in time allows a server to implement access control by restricting or amplifying rights. Languages in which processes can send messages to arbitrary destinations, or in which a process is unable to specify the senders from whom it is willing to receive must generally resort to user-level mechanisms for probabilistic protection (e.g. with keys drawn randomly from a sparse set). Linda [21] is in some sense at the opposite extreme from Lynx. Its *tuple space* mechanism provides the equivalent of a completely-connected communication graph, freeing the programmer from all concern with links or their equivalent. At the same time, however, it sacrifices both message type checking and the access control so important to communication between untrusted programs.

Conversation management

In a language with multiple threads of control, the concurrency between threads can be used for two quite different purposes. It can serve to express true parallelism, for the sake of enhanced performance, or it can serve as a program structuring tool to simplify the exposition of certain kinds of algorithms. In a server process the latter purpose is particularly important; it captures the existence of independent, partially-completed conversations with multiple clients. Unless the hardware permits physical parallelism within an individual server process, the goal of running threads in parallel may not be important at all. It is of course attractive to have a lightweight thread mechanism that addresses both goals at once, but even the appearance of genuine parallelism introduces the need for fine-grained synchronization on data that is shared between threads. In a monitor-based language with a stub generator (e.g. Cedar [22]), the programmer must keep track of two very different forms of synchronization: monitors shared by threads in the same address space and remote procedure calls between threads in different address spaces. Dissatisfaction with a similar approach in Washington's Eden project [15] was a principal motivation for the development of the Emerald language [23]. Emerald provides an object-oriented model that eliminates the distinction between local and remote invocations, but it still requires monitors to synchronize concurrent invocations within a single object. SR [20] takes a different approach to unifying remote and local invocations, but still requires semaphores for certain kinds of local synchronization.

No matter how elegant the synchronization mechanism, its use is still a burden to the programmer. In an implementation without true parallelism, pseudo-concurrent semantics for threads create the appearance of race conditions that should not even exist. They force the use of explicit synchronization on even the most simple operations.† Lynx abandons the possibility of true parallelism within processes in favor of simpler semantics. Threads in Lynx, like coroutines, run until they block. They are purely a program structuring tool.

In either case, whether threads are said to run in parallel or not, a process that is communicating with several peers at once can benefit tremendously from a careful integration of thread management with the facilities for passing messages. Implicit receipt of messages, for example, allows a new thread of control to be created automatically in response to certain kinds of incoming messages, making it easier to associate a separate thread with each conversation with a client. Implicit receipt is characteristic of concurrent languages with stub generators, and is also found in Argus, Emerald, and SR. In Lynx it is extended to permit the creation of threads in a nested lexical context, so that related threads can share state. Because communication results in unpredictable delays, communication statements also constitute an obvious place at which to re-schedule threads. Because a process executes a different thread when the current one sends or receives, Lynx is able to provide the conceptual clarity of remote procedure calls between threads while allowing a process to appear from the outside as if it were using asynchronous, non-blocking messages.

6. IMPLEMENTATION EXPERIENCE

An implementation of Lynx for Wisconsin's Charlotte operating system was completed in 1984. It was ported to a simplified version of Charlotte in 1987. Charlotte runs on a collection of VAXen connected by a token ring [24]. An implementation for the Chrysalis operating system on the BBN Butterfly multiprocessor was completed in 1986. Other designs exist for Unix (using TCP/IP) and for an experimental system known as SODA [25]. Implementation of the Unix design was begun but not completed; the SODA design exists on paper only. An implementation (based on the Chrysalis version) is being developed for the Psyche multiprocessor operating system [26]. In addition to providing a testbed for evaluating Lynx, our implementation experience has led to

†To increment a shared variable, for example, a Lynx thread need not worry about atomicity. It can assume that a context switch will not occur between its read and write. If we pretend that threads are truly parallel, then the program will not be "correct" unless we write code to specify that the physically atomic increment operation should also be semantically atomic.

unexpected insights into the relationship between a language run-time package and the underlying operating system [4, 6], and also into the factors that contribute to message passing overhead [7].

6.1. The language/kernel interface

A distributed operating system provides a process abstraction and primitives for communication between processes. A distributed programming language can regularize the use of the primitives, making them both safer and more convenient. The level of abstraction of the primitives, and therefore the division of labor between the operating system and the language support routines, has serious ramifications for efficiency and flexibility. Lynx is one of the few distributed languages that has been implemented on top of more than one operating system.

Simply put, the implementation experience with Lynx is that the more primitive the operating system (within reason) the easier it is to build a language above it. When we set out to implement Lynx we did not expect to discover this result. Symmetric, two-directional links are directly supported by Charlotte. The original motivation for Lynx was to build a language around them. Yet despite the fact that Charlotte kernel calls provided links as a fundamental abstraction, the implementation of Lynx was extremely complicated and time-consuming. Several of the functions provided by the kernel were almost, but not quite, what the run-time package needed. For example, Charlotte's *receive* function provided no way to say that only reply messages were wanted (and not requests). A complicated protocol was required in the run-time package in order to reject and return unwanted requests. Similarly, the Charlotte *send* function allows links to be enclosed in messages, but only one at a time. Additional run-time protocol was required to packetize multi-link messages.

By comparison, implementation of Lynx on top of Chrysalis was surprisingly easy, despite the fact that Chrysalis has no notion of a link or even of a message. What Chrysalis *does* provide are low-level facilities for creating shared memory blocks and for atomically manipulating flags and queues. The following section explains how links can be built from these primitives. The fact that Chrysalis supports shared memory is a significant but not deciding factor in it suitability for Lynx. Our paper implementation for the message-based primitives of SODA is equally simple. Our TCP/IP design lies somewhere in the middle.

Like most distributed operating systems, Charlotte was designed with the expectation that programmers would invoke its primitives directly. This expectation appears to have been naive, but by no means unique. The proliferation of remote procedure call stub generators suggests that users of a wide range of message-passing operating systems have found their primitives too primitive to use. Unfortunately, the creation of interfaces that are *almost* usable for day-to-day programming has meant that substantial amounts of functionality and, consequently, flexibility, have been hidden from the user. Remote procedure calls may work, but alternative approaches to naming, buffering, synchronization, error recovery, or flow control are generally not available.

Our experience with Lynx suggests that an operating system kernel should either be designed to support a single high-level language (as, for example, in the dedicated implementations of Argus [27], Linda [28], and SR [20]), or else should provide only the lowest common denominator for things that will be built upon it. A middle-level interface is likely to be both awkward and slow: awkward because it has sacrificed the flexibility of the more primitive system; slow because it has sacrificed its simplicity. We recommend low-level kernels because they can maintain flexibility without introducing costs.

6.2. The significance of hints

Within the Charlotte kernel, the implementation of link movement was a major source of complexity. A decision was made early in the design process that the kernel at each end of a link would know the location of the other end at all times. An efficient, symmetric protocol for moving links was eventually devised, but it was surprisingly subtle. It is possible for both ends of a link to be in motion at once, and the interaction of link movement with link destruction, message transmission, and the cancellation of send and receive requests complicates the situation further.

In the Lynx implementation designed for SODA, processes keep only hints regarding the locations of the other ends of links. When moving a link end, a process sends an update notice to the suspected location of the other end, informing it of the move. With both ends of a link in

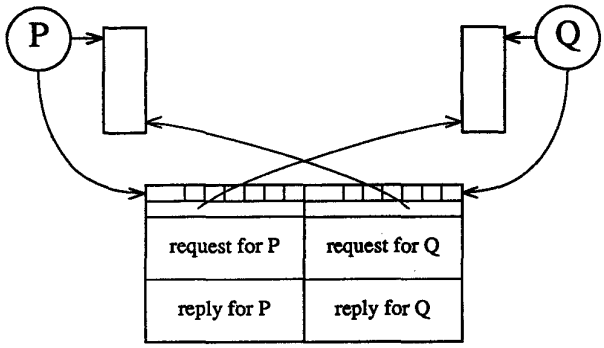


Fig. 5. Chrysalis link implementation.

motion at once, update notices can be lost. A process that discovers its hint to be incorrect must resort to a broadcast mechanism for link-end location discovery. The resulting protocol is substantially simpler than the one employed in Charlotte.

Previous papers have cited this experience as evidence that "hints can be better than absolutes" [6]. The caveat in this lesson is that the correctness of the overall algorithm must not *depend* on the hints. If the work they facilitate is important, it must be possible to detect when they fail, and there must be a fallback mechanism that can be counted on to work.

A Lynx implementation on SODA could employ two levels of fallback, one utilizing unreliable hardware broadcast, the other a reliable but expensive broadcast protocol. In a Lynx implementation for Unix, operation over potentially long-haul TCP connections would preclude the use of broadcast. Our Unix design therefore employs the Charlotte method of absolutes. It creates a single TCP connection between every pair of processes that share one or more links. A handshaking protocol implemented on top of this connection multiplexes as many links as necessary, moves their ends when required, and implements the request/reply message screening that was unavailable in Charlotte. Send and receive requests are never cancelled. The basic request-reply protocol employs 8 different message types, including negative and positive acknowledgments (This is the same set of messages types used in the Chrysalis implementation, described in the following section.) Link movement requires 3 more messages types, one of which is used only when passing a link to a new child process as a command-line argument. The link movement protocol employs 4 states and 9 transitions; a link end can be valid, invalid, in transit, or "floating", which means that the link is moving at both ends.

6.3. The Chrysalis and Psyche implementations

The Chrysalis implementation of Lynx consists of a cross compiler that runs on a host machine and a run-time support package that implements links in terms of Chrysalis primitives. For compatibility reasons, and to simplify the implementation, the compiler generates C for "intermediate code". Errors in the Lynx source inhibit code generation, so the output, if any, will pass through the C compiler without complaint. Programmers are in general unaware of the C back end. The compiler is internally conventional, comprising approx. 18000 lines of Pascal. The run-time support package comprises 5600 lines of C and 300 lines of assembler, the bulk of which is devoted to message passing, thread management, and exception handling.

Special measures are required for stack management, to support the sharing of non-global data among threads. One approach would be to place each trunk of a cactus stack in single, contiguous array, allocated when its thread is created. This would require, however, that the space needs of a thread be known in advance, or estimated liberally, with considerable waste. An attractive alternative is to allocate each individual activation frame dynamically. A caching strategy for frames makes allocation relatively inexpensive, but further optimization is still desirable. As it parses its input files, the Lynx compiler keeps track of which subroutines contain statements that may cause a thread switch. Space for such routines must be allocated in the cactus stack, since returns needs not occur in LIFO order. For routines that *cannot* cause a thread switch, however, space may be allocated on an ordinary stack. In practice most subroutines (and particularly those

that are frequently called) can be seen to be sequential. The coroutine-like semantics of threads in Lynx allow these routines to be implemented at precisely the same cost as in conventional sequential languages.

For message passing, every process in the Chrysalis implementation allocates an atomic queue (provided by the operating system) when it first begins execution. This queue is used to receive notifications of messages sent and received on any of the process's links. A link is represented by a block of shared memory, mapped into the address spaces of the two connected processes (see Fig. 5). The shared memory object contains buffer space for a single request and a single reply in each direction. Since dynamic allocation and re-mapping of message buffers would be prohibitively expensive, messages are limited to a fixed maximum length, currently 2000 bytes. Each process keeps an internal list of the threads that are waiting for buffers to be emptied or filled.

In addition to message buffers, each link object also contains a set of flag bits and the names of the atomic queues for the processes at each end of the link. When a process gathers a message into a buffer or scatters a message out of a buffer into local variables, it sets a flag in the link object (atomically) and then enqueues a notice of its activity on the atomic queue for the process at the other end of the link. When all of the process's threads are blocked, it attempts to dequeue a notice from its own atomic queue. The Chrysalis kernel, which implements queue operations, blocks the process if the queue is empty and unblocks it when the first new notice is enqueued.

The flag bits permit the implementation of link movement. Whenever a process dequeues a notice from its atomic queue it checks to see that it owns the mentioned link end and that the appropriate flag is set in the corresponding object. If either check fails, the notice is discarded. Every change to a flag is eventually reflected by a notice on the appropriate atomic queue, but not every queue notice reflects a change to a flag. A link is moved by passing the (address-space-independent) name of its memory object in a message. When the message is received, the sending process removes the memory object from its address space. The receiving process maps the object *into* its address space, changes the information in the object to name its own atomic queue, and then inspects the flags. It enqueues notices on its own queue for any of the flags that are set.

The Psyche implementation of Lynx, currently in the planning stage, will resemble the Chrysalis version. The principal difference is that instead of using atomic queues, the Psyche version will rely on a *protected procedure call* mechanism that allows a process to invoke protocol operations in the run-time package of a communication partner directly, rather than enqueuing a notice indicating that the partner should perform the operation itself. If the partner process is blocked pending completion of a communication request, the protected procedure call mechanism will serve to unblock it.

The principal advantage of this approach is that some of the work currently performed by the partner that reaches a rendezvous last (e.g. sends a message that is already wanted, or receives a message that was already sent) can in Psyche be performed by the partner that reaches the rendezvous first. This change should serve to reduce the latency of many message transfers. It does not constitute a security loophole because operations invoked in the run-time support of a communication partner execute in the partner's address space, using the partner's code.

6.4. The cost of message-passing

In our Butterfly implementation of Lynx, the simplest remote operations complete in <2 ms. To place this figure in perspective, a call to an empty procedure takes $10\,\mu$s on an individual Butterfly node. An atomic test-and-set operation on remote memory takes $35\,\mu$s. An atomic enqueue or dequeue operation takes $80\,\mu$s. In the following table, nullop is a trivial remote operation with no parameters. Bigop is the same as nullop, but includes 1000 bytes of parameters in each direction. Explicit receipt uses an accept statement; implicit receipt uses a binding to an entry.

	Explicit receipt: process nodes		Implicit receipt: process nodes	
	different	same	different	same
nullop	1.80 ms	2.58 ms	2.04 ms	2.76 ms
bigop	3.45 ms	4.21 ms	3.72 ms	4.42 ms

21% – actual communication
Clearing and setting flag bits (12.4), posting notices on queues (7.3), calculating locations of message buffers (1.0).

22% – thread management
Thread queue management (4.8), queue searching (dispatcher) (3.1), context switches (6.4), cactus stack frame allocation (6.8), buffer acquisition (1.3).

11% – bookkeeping
Keeping track of which threads want which sorts of services and which threads are willing to provide them.

18% – checking and exception handling
Verifying link validity (3.2), verifying success of kernel calls (9.2), type checking (0.8), establishment of Lynx exception handlers (4.4), initialization of stack frame exception information (0.6).

6% – protocol option testing
Checking for link movement (2.0), asynchronous notifications (1.0), premature requests (1.6), optional acknowledgments (1.8).

22% – miscellaneous overhead
Timing loop overhead (0.6), dispatcher loop and case statement overhead (1.4), procedure-call linkage (14.7), caching of constants in registers (5.6).

Fig. 6. Contributors to message-passing overhead (in % of total work performed).

Inter-node operations finish more quickly than intra-node operations because the two processors can overlap their computations. Implicit receipt costs more than explicit receipt because of the need to create and destroy a thread. We have reason to believe that these times could be reduced by additional tuning, but it seems unlikely that the lowest figure would drop below a millisecond and a half. A remote invocation is thus two orders of magnitude more expensive than a local procedure call, a result that is consistent with most other well-tuned message-passing systems.

Like many researchers, we found the cost of message passing to be both frustrating and puzzling. Not only did we wish that things worked faster, we also did not *understand* why they worked at the speed they did. In order to obtain a better explanation of "where the time goes", we profiled benchmark programs at the instruction level and assigned each individual instruction to one of 23 different functional categories. The results of this profiling are summarized in Fig. 6. A timeline for a 2.0 ms remote operation appears in Fig. 7. The timeline indicates the amount of time devoted to each of the phases of a remote invocation, but provides relatively little insight into the expense of individual language features involved in message passing.

Procedure call overhead and flag bit manipulation are the only single items in the profiling table that account for more than 10% of the total communication overhead. Small savings could undoubtedly be realized here and there, but there does not seem to be any way to achieve significant performance gains without eliminating language features. Work by other researchers tends to confirm the hypothesis that data transmission times do not dominate the cost of practical message-passing systems [29–32]. High-level semantic functions such as addressing, dispatching, bookkeeping, testing and error handling are at least as significant, and often more so. One millisecond appears to be a nearly universal lower bound on round-trip communication times with mid-1980s microprocessor-based architectures, suggesting that it may be extremely difficult to provide attractive message-passing semantics in significantly fewer than 1000 instructions.

Recent implementations of lightweight remote procedure call [33, 34] have broken the 1000 instruction barrier decisively by pre-computing significant portions of the invocation mechanism during an explicit connect-to-service operation. This technique requires extensive kernel support, however, and does not generalize in any obvious way to languages that must be implemented on a general-purpose operating system, or that lack a well-defined concept of connecting to a service.

7. SUMMARY AND CONCLUSION

Lynx is a programming language providing convenient, typesafe message passing among application and server processes in a distributed environment. Numerous programs have been written in Lynx over the course of the past 5 years, both as research projects and as coursework

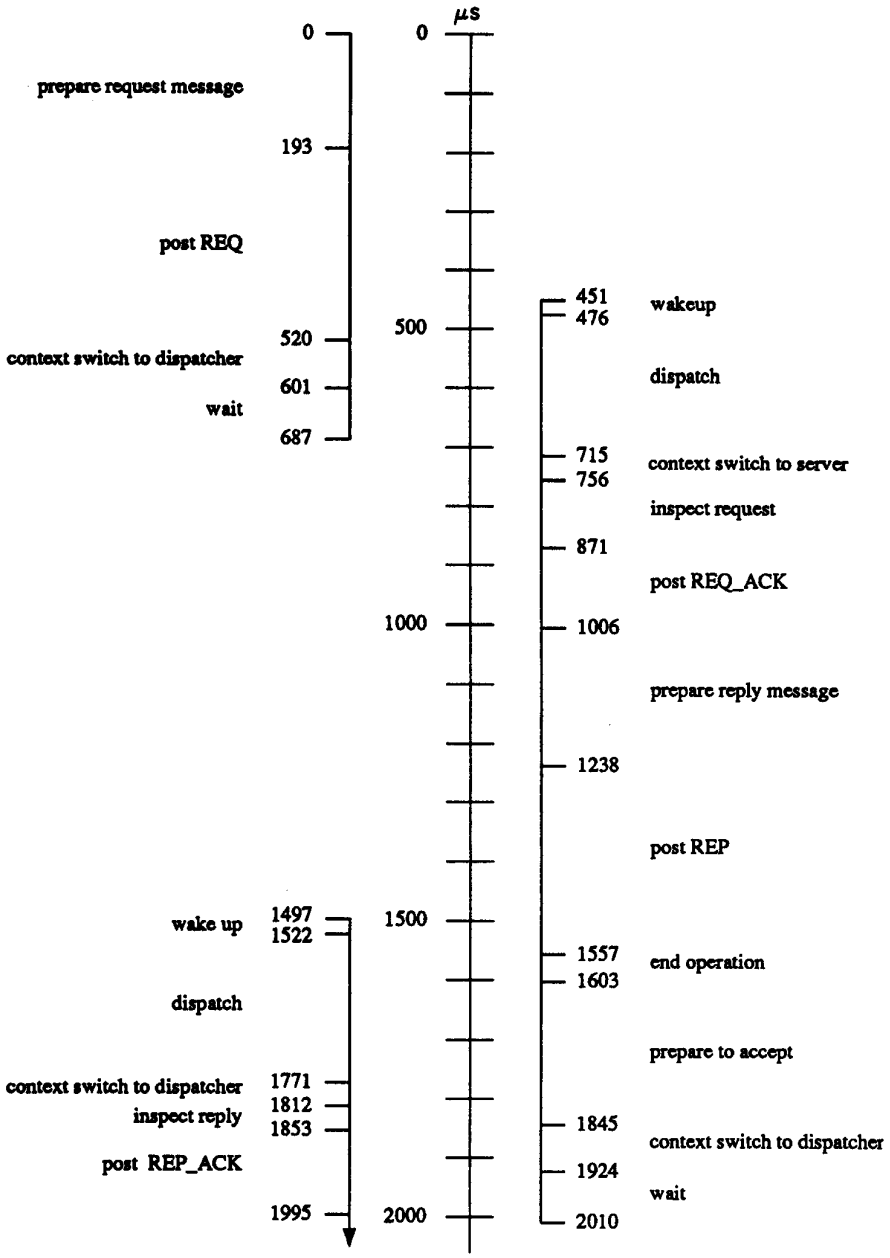


Fig. 7. Timeline for a simple remote invocation.

[35, 36]. In comparison to programs that perform communication through library routines, Lynx programs are consistently shorter, easier to debug, easier to write, and easier to read. Much of the explanation is simply the difference between a language and the lack thereof; Ada, Argus, Linda, NIL, and SR can make similar claims. With a few exceptions, all these languages provide attractive syntax, secure type checking, error handling with exceptions, and automatic management of context for multiple conversations.

Beyond these facilities, however, Lynx provides an unusual degree of run-time flexibility.

- Symmetric communication links provide abstraction and transparent reconfiguration without the restrictions of compile-time type checking.

- The ability to distinguish between clients provides access control and protection.
- Mutually-exclusive threads provide context for multiple conversations without the complexity of synchronizing access to memory. The integration of threads with communication combines the conceptual clarity of remote procedure calls with the performance of non-blocking messages.

Each of these advantages can be of use in single-program applications. More important, however, they extend the advantages of language support to autonomous but interacting programs.

Experience with several implementations of Lynx suggests that the interface between a distributed operating system and a distributed programming language should either be very low level (close to the hardware) or very high level (close to the language). Use of anything in between is likely to be both cumbersome and slow. Instrumentation of the Lynx implementation for the BBN Butterfly multiprocessor indicates that message passing owes its expense to the confluence of many smaller costs, with no single aspect of communication predominating.

Acknowledgements—Raphael Finkel supervised the Ph.D. thesis in which Lynx was originally defined. Marvin Solomon provided additional guidance. Ken Yap helped to port Lynx to the Butterfly. Alan Cox assisted with performance studies. Tom Virgilio ported Lynx to the simplified version of Charlotte. Bill Bolosky helped design the TCP/IP implementation, and implemented most of it. The link movement protocol is based on ideas of Bryan Rosenburg and Bill Kalsow. Numerous students, both at Wisconsin and at Rochester, provided feedback on the language and compiler. I am grateful to Lawrence Crowl and to one particularly thorough referee, whose comments improved this paper substantially. At the University of Wisconsin, this work was supported in part by NSF CER grant number MCS-8105904, DARPA contract number N0014-82-C-2087, and a Bell Telephone Laboratories Doctoral Scholarship. At the University of Rochester, this work was supported in part by NSF CER grant number DCR-8320136, DARPA/ETL contract number DACA76-85-C-0001, and an IBM Faculty Development Award.

REFERENCES

1. Scott, M. L. Language support for loosely-coupled distributed programs. *IEEE Trans. Software Engng* **SE-13:** 88–103; 1987.
2. Scott, M. L. LYNX reference manual. BPR 7, Computer Science Department, University of Rochester; 1986 (revised).
3. Artsy, Y., Chang, H. and Finkel, R. Interprocess communication in Charlotte. *IEEE Software* **4:** 22–28; 1987.
4. Finkel, R. A., Scott M. L., Artsy Y. and Chang, H. Experience with Charlotte: simplicity and function in a distributed operating system. *IEEE Trans. Software Engng* **15:** 676–685; 1989. Extended abstract presented at the *IEEE Workshop on Design Principles for Experimental Distributed Systems*, Purdue University; 1986.
5. Scott, M. L. and Finkel, R. A. A simple mechanism for type security across-compilation units. *IEEE Trans. Software Engng* **14:** 1238–1239; 1988.
6. Scott, M. L. The interface between distributed operating system and high-level programming language. *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 242–249; 1986.
7. Scott, M. L. and Cox, A. L. An empirical study of message-passing overhead. *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pp. 536–543; 1987.
8. Wirth, N. Modula: a language for modular multiprogramming. *Software—Practice and Experience* **7:** 3–35; 1977.
9. Birrell, A. D. and Nelson B. J. Implementing remote procedure calls. *ACM Trans. Comput. Syst.* **2:** 39–59; 1984. Originally presented at the *Ninth ACM Symposium on Operating Systems Principles*; 1983. Originally presented at the *Ninth ACM Symposium on Operating Systems Principles*; 1983.
10. Jones, M. B., Rashid R. F. and Thompson, M. R. Matchmaker: an interface specification language for distributed processing. *Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages*, pp. 225–235; 1985.
11. Scott, M. L. A framework for the evaluation of high-level languages for distributed computing. Computer Sciences Technical Report # 563, University of Wisconsin–Madison; 1984.
12. Welsh, J. and Lister, A. A comparative study of task communication in Ada. *Software—Practice and Experience* **11:** 257–290; 1981.
13. Hensgen, D. and Finkel, R. Dynamic server squads in Yackos. *Proceedings of the First Workshop on Experiences with Building Distributed and Multiprocessor Systems*, pp. 73–89; 1989.
14. Fishburn, J. P. An analysis of speedup in parallel algorithms. Ph.D. thesis, Computer Sciences Technical Report # 431, University of Wisconsin–Madison; 1981.
15. Black, A. P. Supporting distributed applications: experience with Eden. *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pp. 181–193; 1985. In *ACM Oper. Syst. Rev.* **19:** 5.
16. U.S. Department of Defense. Reference Manual for the Ada® Programming Language. Available as *Lecture Notes in Computer Science # 106.* New York: Springer; 1981.
17. Strom, R. E. and Yemini, S. 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; 1983. In *ACM SIGPLAN Notices* **18:** 6.
18. Liskov, B. and Scheifler R. Guardians and actions: linguistic support for robust, distributed programs. *ACM Trans. Prog. Lang. Syst.* **5:** 381–404; 1983.
19. Tichy, W. F. Smart recompilation. *ACM Trans. Prog. Lang. Syst.* **8:** 273–291; 1986. Relevant correspondence appears in Vol. 10, No. 4.

20. Andrews, G. R., Olsson, R. A., Coffin, M., Elshoff, I. J. P., Nilsen, K., Purdin, T. and Townsend, G. An overview of the SR language and implementation. *ACM Trans. Prog. Lang. Syst.* **10:** 51–86; 1988.
21. Gelernter, D. Generative communication in Linda. *ACM Trans. Prog. Lang. Syst.* **7:** 80–112; 1985.
22. Swinehart, D., Zellweger, P., Beach, R. and Hagmann, R. A structural view of the Cedar programming environment. *ACM Trans. Prog. Lang. Syst.* **8,** 419–490; 1986.
23. Black, A., Hutchinson, N., Jul, E. and Levy, H. Object structure in the Emerald system. *OOPSLA '86 Conference Proceedings*, pp. 78–86; 1986. In *ACM SIGPLAN Notices* **21:** 11.
24. DeWitt D. J., Finkel, R. and Solomon, M. The Crystal multicomputer: design and implementation experience. *IEEE Trans. Software Engng* **SE-13:** 953–966; 1987.
25. Kepecs, J. and Solomon, M. SODA: A simplified operating system for distributed applications. *ACM Oper. Syst. Rev.* **19:** 45–56; 1985. Originally presented at the *Third Annual ACM Symposium on Principles of Distributed Computing*; 1984.
26. Scott, M. L., LeBlanc, T. J. and Marsh, B. D. Multi-model parallel programming in Psyche. *Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming*, pp. 70–78; 1990. In *ACM SIGPLAN Notices* **25:** 3.
27. Liskov, B., Curtis, D., Johnson, P. and Scheifler, R. Implementation of Argus. *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pp. 111–122; 1987. In *ACM Oper. Syst. Rev.* **21:** 5.
28. Carriero, N. and Gelernter, D. The S/Net's Linda kernel. *ACM Trans. Comput. Syst.* **4:** 110–129; 1986. Originally presented at the *Tenth ACM Symposium on Operating Systems Principles*; 1985.
29. Cheriton, D. R. and Zwaenepoel, W. The distributed V kernel and its performance for diskless workstations. *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pp. 129–140; 1983. In *ACM Oper. Syst. Rev.* **17:** 5.
30. LeBlanc, T. J. and Cook, R. P. An analysis of language models for high-performance communication in local-area networks. *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, pp. 65–72; 1983. In *ACM SIGPLAN Notices* **18:** 6.
31. Nelson, B. J. Remote procedure call. Ph.D. thesis, Technical Report CMU-CS-81-119, Carnegie-Mellon University; 1981.
32. Spector, A. Z. Performing remote operations efficiently on a local computer network. *Commun. ACM* **25:** 246–260; 1982.
33. Bershad, B. N., Anderson, T. E., Lazowska, E. D. and Levy, H. M. Lightweight remote procedure call. *ACM Trans. Comput. Syst.* **8:** 37–55; 1990. Originally presented at the *Twelfth ACM Symposium on Operating Systems Principles*; 1989.
34. Schroeder, M. and Burrows, M. Performance of Firefly RPC. *ACM Trans. Comput. Syst.* **8:** 1–17; 1990. Originally presented at the *Twelfth ACM Symposium on Operating Systems Principles*; 1989.
35. Brown C. M., Fowler, R. J., LeBlanc T. J., Scott, M. L., Srinivas, M., *et al.* DARPA parallel architecture benchmark study. BRP 13, Computer Science Department, University of Rochester; 1986.
36. Finkel, R. *et al.* Experience with Crystal, Charlotte, and LYNX. Computer Sciences Technical Reports # 630, # 649, and # 673, University of Wisconsin–Madison; 1986.

About the Author—Michael L. Scott is an Assistant Professor of Computer Science at the University of Rochester. He received his Ph.D. in computer sciences from the University of Wisconsin–Madison in 1985. His research focuses on programming languages, operating systems, and program development tools for parallel and distributed computing. He is co-leader of Rochester's Psyche parallel operating system project, and is the recipient of a 1986 IBM Faculty Development Award.