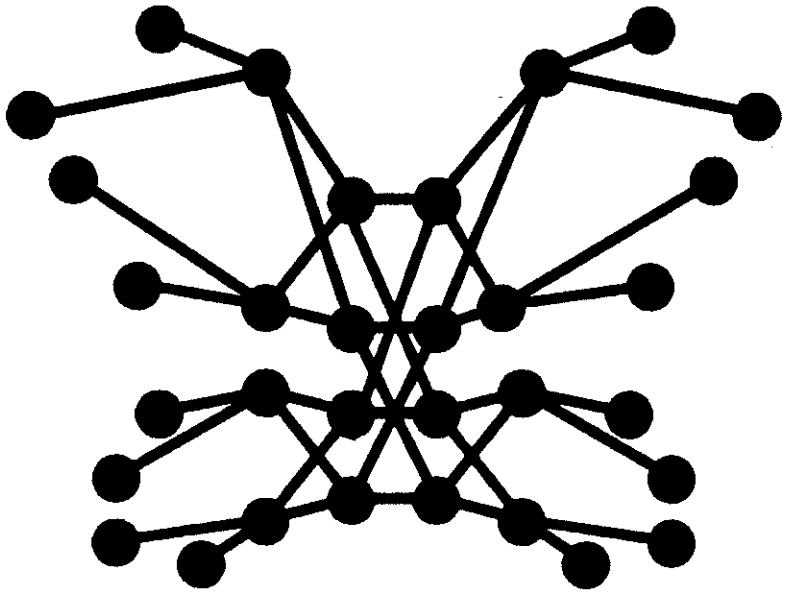


Butterfly Project Report
6

The Interface Between Distributed Operating System and High-Level Programming Language

Michael L. Scott

(revised) September 1986

Computer Science Department
University of Rochester
Rochester, NY 14627

The Interface Between Distributed Operating System and High-Level Programming Language

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

TR 182 (revised)
September 1986

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

This paper was presented at the *1986 International Conference on Parallel Processing*, St. Charles, IL, 20 August 1986.

The Interface Between Distributed Operating System and High-Level Programming Language

Michael L. Scott

Computer Science Department
University of Rochester
Rochester, NY 14627

ABSTRACT

A distributed operating system provides a process abstraction and primitives for communication between processes. A distributed programming language regularizes 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. Experience with three implementations of the LYNX distributed programming language suggests that functions that can be implemented on either side of the interface are best left to the language run-time package.

1. Introduction

Recent years have seen the development of a large number of distributed programming languages and an equally large number of distributed operating systems. While there are exceptions to the rule, it is generally true that individual research groups have focused on a single language, a single operating system, or a single language/O.S. pair. Relatively little attention has been devoted to the relationship between languages and O.S. kernels in a distributed setting.

Amoeba [16], Demos-MP [17], Locus [27], and the V kernel [8] are among the better-known distributed operating systems. Each by-passes language issues by relying on a simple library-routine interface to kernel communication primitives. Eden [6] and Cedar [25] have both devoted a considerable amount of attention to programming language issues, but each is very much a single-language system. The Accent project at CMU [18] is perhaps the only well-known effort to support more than one programming language on a single underlying kernel. Even so, Accent is only able to achieve its multi-lingual character by insisting on a single, universal model of interprocess communication based on remote procedure calls [12]. Languages with other models of process interaction are not considered.

In the language community, it is unusual to find implementations of the same distributed programming language for more than one operating system, or indeed for any *existing* operating system. Dedicated, special-purpose kernels are under construction for Argus [15], SR [1, 2], and NIL [23, 24]. Several dedicated implementations have been designed for Linda [7, 11]. No distributed implementations have yet appeared for Ada [26].

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

This paper was presented at the *1986 International Conference on Parallel Processing*, St. Charles, IL, 20 August 1986.

If parallel or distributed hardware is to be used for *general-purpose* computing, we must eventually learn how to support multiple languages efficiently on a single operating system. Toward that end, it is worth considering the division of labor between the language run-time package and the underlying kernel. Which functions belong on which side of the interface? What is the appropriate level of abstraction for universal primitives? Answers to these questions will depend in large part on experience with a variety of language/O. S. pairs.

This paper reports on implementations of the LYNX distributed programming language for three existing, but radically different, distributed operating systems. To the surprise of the implementors, the implementation effort turned out to be substantially easier for kernels with low-level primitives. If confirmed by similar results with other languages, the lessons provided by work on LYNX should be of considerable value in the design of future systems.

The first implementation of LYNX was constructed during 1983 and 1984 at the University of Wisconsin, where it runs under the Charlotte distributed operating system [4, 10] on the Crystal multicomputer [9]. The second implementation was designed, but never actually built, for Kepecs and Solomon's SODA [13, 14]. A third implementation has recently been released at the University of Rochester, where it runs on BBN Butterfly multiprocessors [5] under the Chrysalis operating system.

Section 2 of this paper summarizes the features of LYNX that have an impact on the services needed from a distributed operating system kernel. Sections 3, 4, and 5 describe the three LYNX implementations, comparing them one to the other. The final section discusses possible lessons to be learned from the comparison.

2. LYNX Overview

The LYNX programming language is not itself the subject of this article. Language features and their rationale are described in detail elsewhere [20, 21, 22]. For present purposes, it suffices to say that LYNX was designed to support the loosely-coupled style of programming encouraged by a distributed operating system. Unlike most existing languages, LYNX extends the advantages of high-level communication facilities to processes designed in isolation, and compiled and loaded at disparate times. LYNX supports interaction not only between the pieces of a multi-process application, but also between separate applications and between user programs and long-lived system servers.

Processes in LYNX execute in parallel, possibly on separate processors. There is no provision for shared memory. Interprocess communication uses a mechanism similar to remote procedure calls (RPC), on virtual circuits called **links**. Links are two-directional and have a single process at each end. Each process may be divided into an arbitrary number of threads of control, but the threads execute in mutual exclusion and may be managed by the language run-time package, much like the coroutines of Modula-2 [28].

2.1. Communication Characteristics

(The following paragraphs describe the communication behavior of LYNX processes. The description does not provide much insight into the way that LYNX programmers think about their programs. The intent is to describe the externally-visible characteristics of a process that must be supported by kernel primities.)

Messages in LYNX are not received asynchronously. They are queued instead, on a link-by-link basis. Each link end has one queue for incoming requests and another for incoming replies. Messages are received from a queue only when the queue is **open** and the process that owns its end has reached a well-defined **block point**. Request queues may be opened or closed under explicit process control. Reply queues are opened when a request has been sent and a reply is expected. The set of open queues may therefore vary from one block point to the next.

A blocked process waits until one of its previously-sent messages has been received, or until an incoming message is available in at least one of its open queues. In the latter case, the process

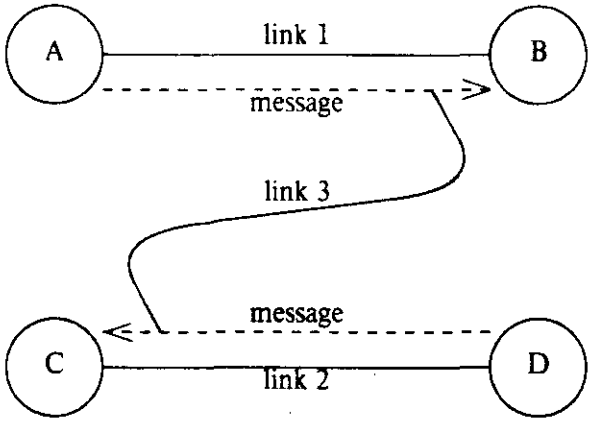


figure 1: link moving at both ends

chooses a non-empty queue, receives that queue's first message, and executes through to the next block point. For the sake of fairness, an implementation must guarantee that no queue is ignored forever.

Messages in the same queue are received in the order sent. Each message blocks the sending coroutine within the sending process. The process must be notified when messages are received in order to unblock appropriate coroutines. It is therefore possible for an implementation to rely upon a stop-and-wait protocol with no actual buffering of messages in transit. Request and reply queues can be implemented by lists of blocked coroutines in the run-time package for each sending process.

The most challenging feature of links, from an implementor's point of view, is the provision for *moving* their ends. Any message, request or reply, can contain references to an arbitrary number of link ends. Language semantics specify that receipt of such a message has the side effect of moving the specified ends from the sending process to the receiver. The process at the far end of each moved link must be oblivious to the move, even if it is currently relocating its end as well. In figure 1, for example, processes A and D are moving their ends of link 3, independently, in such a way that what used to connect A to D will now connect B to C.

It is best to think of a link as a flexible hose. A message put in one end will eventually be delivered to whatever process happens to be at the other end. The queues of available but unreceived messages for each end are associated with the link itself, not with any process. A moved link may therefore (logically at least) have messages inside, waiting to be received at the moving end. In keeping with the comment above about stop-and-wait protocols, and to prevent complete anarchy, a process is not permitted to move a link on which it has *sent* unreceived messages, or on which it owes a reply for an already-received request.

2.2. Kernel Requirements

To permit an implementation of LYNX, an operating system kernel must provide processes, communication primitives, and a naming mechanism that can be used to build links. The major questions for the designer are then 1) how are links to be represented? and 2) how are RPC-style request and reply messages to be transmitted on those links? It must be possible to move links

without losing messages. In addition, the termination of a process must destroy all the links attached to that process. Any attempt to send or receive a message on a link that has been destroyed must fail in a way that can be reflected back into the user program as a run-time exception.

3. The Charlotte Implementation

3.1. Overview of Charlotte

Charlotte [4, 10] runs on the Crystal multicomputer [9], a collection of 20 VAX 11/750 **node** machines connected by a 10-Mbit/second token ring from Proteon Corporation.

The Charlotte **kernel** is replicated on each node. It provides direct support for both **processes** and **links**. Charlotte links were the original motivation for the circuit abstraction in LYNX. As in the language, Charlotte links are two directional, with a single process at each end. As in the language, Charlotte links can be created, destroyed, and moved from one process to another. Charlotte even guarantees that process termination destroys all of the process's links. It was originally expected that the implementation of LYNX-style interprocess communication would be almost trivial. As described in the rest of this section, that expectation turned out to be naive.

Kernel calls in Charlotte include the following:

MakeLink (var end1, end2 : link)
Create a link and return references to its ends.

Destroy (myend : link)
Destroy the link with a given end.

Send (L : link; buffer : address; length : integer; enclosure : link)
Start a **send activity** on a given link end, optionally enclosing one end of some other link.

Receive (L : link; buffer : address; length : integer)
Start a **receive activity** on a given link end.

Cancel (L : link; d : direction)
Attempt to cancel a previously-started send or receive activity.

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

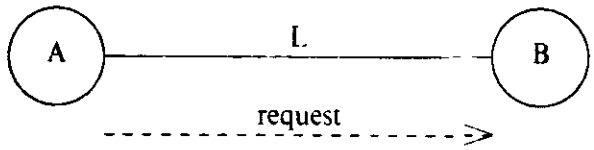
All calls return a status code. All but *Wait* are guaranteed to complete in a bounded amount of time. *Wait* blocks the caller until an activity completes.

The Charlotte kernel matches send and receive activities. It allows only one outstanding activity in each direction on a given end of a link. Completion must be reported by *Wait* before another similar activity can be started.

3.2. Implementation of LYNX

The language run-time package represents every LYNX link with a Charlotte link. It uses the activities of the Charlotte kernel to simulate the request and reply queues described in section 2.1. It starts a send activity on a link whenever a process attempts to send a request or reply message. It starts a receive activity on a link when the corresponding request or reply queue is opened, if both were closed before. It attempts to cancel a previous-started receive activity when a process closes its request queue, if the reply queue is also closed. The multiplexing of request and reply queues onto receive activities was a major source of problems for the implementation effort. A second source of problems was the inability to enclose more than one link in a single Charlotte message.

3.2.1. Screening Messages For the vast majority of remote operations, only two Charlotte messages are required: one for the request and one for the reply. Complications arise, however, in a number of special cases. Suppose that process A requests a remote operation on link L.



Process B receives the request and begins serving the operation. A now expects a reply on L and starts a receive activity with the kernel. Now suppose that before replying B requests another operation on L, in the reverse direction (the coroutine mechanism mentioned in section 2 makes such a scenario entirely plausible). A will receive B's request before the reply it wanted. Since A may not be willing to serve requests on L at this point in time (its request queue is closed). B is not able to assume that its request is being served simply because A has received it.

A similar problem arises if A opens its request queue and then closes it again, before reaching a block point. In the interests of concurrency, the run-time support routines will have posted a *Receive* with the kernel as soon as the queue was opened. When the queue is closed, they will attempt to cancel the *Receive*. If B has requested an operation in the meantime, the *Cancel* will fail. The next time A's run-time package calls *Wait*, it will obtain notification of the request from B, a message it does not want. Delaying the start of receive activities until a block point does not help. A must still start activities for *all* its open queues. It will continue execution after a message is received from exactly *one* of those queues. Before reaching the *next* block point, it may change the set of messages it is willing to receive.

It is tempting to let A buffer unwanted messages until it is again willing to receive from B, but such a solution is impossible for two reasons. First, the occurrence of exceptions in LYNX can require A to cancel an outstanding *Send* on L. If B has already received the message (inadvertently) and is buffering it internally, the *Cancel* cannot succeed. Second, the scenario in which A receives a request but wants a reply can be repeated an arbitrary number of times, and A cannot be expected to provide an arbitrary amount of buffer space.

A must return unwanted messages to B. In addition to the **request** and **reply** messages needed in simple situations, the implementation now requires a **retry** message. *Retry* is a negative acknowledgment. It can be used in the second scenario above, when A has closed its request queue after receiving an unwanted message. Since A will have no *Receive* outstanding, the re-sent message from B will be delayed by the kernel until the queue is re-opened.

In the first scenario, unfortunately, A will still have a *Receive* posted for the reply it wants from B. If A simply returned requests to B in retry messages, it might be subjected to an arbitrary number of retransmissions. To prevent these retransmissions we must introduce the **forbid** and **allow** messages. *Forbid* denies a process the right to send requests (it is still free to send replies). *Allow* restores that right. *Retry* is equivalent to *forbid* followed by *allow*. It can be considered an optimization for use in cases where no replies are expected, so retransmitted requests will be delayed by the kernel.

Both *forbid* and *retry* return any link end that was enclosed in the unwanted message. A process that has received a forbid message keeps a *Receive* posted on the link in hopes of receiving an allow message.[1] A process that has sent a forbid message remembers that it has done so and sends an allow message as soon as it is either willing to receive requests (its request queue is open) or has no *Receive* outstanding (so the kernel will delay all messages).

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

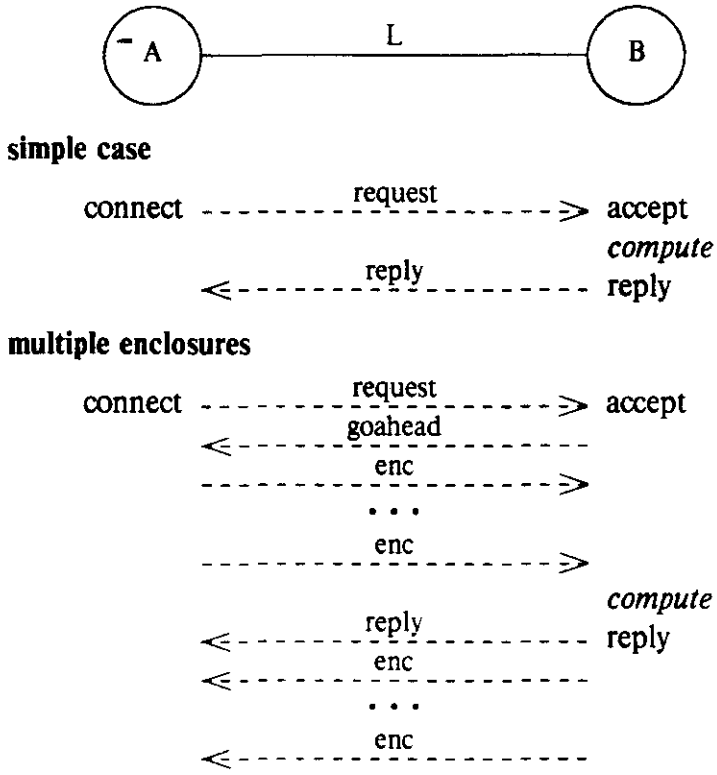


figure 2: link enclosure protocol

3.2.2. Moving Multiple Links To move more than one link end with a single LYNX message, a request or reply must be broken into several Charlotte messages. The first packet contains non-link data, together with the first enclosure. Additional enclosures are passed in empty **enc** messages (see figure 2). For requests, the receiver must return an explicit **goahead** message after the first packet so the sender can tell that the request is wanted. No *goahead* is needed for requests with zero or one enclosures, and none is needed for replies, since a reply is always wanted.

One consequence of packetizing LYNX messages is that links enclosed in unsuccessful messages may be lost. Consider the following chain of events:

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

b) **B** receives the request unintentionally; inspection of the code allows one to prove that only replies were wanted.

c) The sending coroutine in A feels an exception, aborting the request.

d) **B** crashes before it can send the enclosure back to A in a forbid message. From the point of view of language semantics, the message to **B** was never sent, yet the enclosure has been lost. Under such circumstances the Charlotte implementation cannot conform to the language reference manual.

The Charlotte implementation also disagrees with the language definition when a coroutine that is waiting for a reply message is aborted by a local exception. On the other end of the link the

server should feel an exception when *it* attempts to send a no-longer-wanted reply. Such exceptions are not provided under Charlotte because they would require a final, top-level acknowledgment for reply messages, increasing message traffic by 50%.

3.3. Measurements

The language run-time package for Charlotte consists of just over 4000 lines of C and 200 lines of VAX assembler, compiling to about 21K of object code and data. Of this total, approximately 45% is devoted to the communication routines that interact with the Charlotte kernel, including perhaps 5K for unwanted messages and multiple enclosures. Much of this space could be saved with a more appropriate kernel interface.

A simple remote operation (no enclosures) requires approximately 57 ms with no data transfer and about 65 ms with 1000 bytes of parameters in both directions. C programs that make the same series of kernel calls require 55 and 60 ms, respectively. In addition to being rather slow, the Charlotte kernel is highly sensitive to the ordering of kernel calls and to the interleaving of calls by independent processes. Performance figures should therefore be regarded as suggestive, not definitive. The difference in timings between LYNX and C programs is due to efforts on the part of the run-time package to gather and scatter parameters, block and unblock coroutines, establish default exception handlers, enforce flow control, perform type checking, update tables for enclosed links, and make sure the links are valid.

4. The SODA Implementation

4.1. Overview of SODA

As part of his Ph. D. research [13, 14], Jonathan Kepecs set out to design a minimal kernel for a multicomputer. His "Simplified Operating system for Distributed Applications" might better be described as a communications protocol for use on a broadcast medium with a very large number of heterogeneous nodes.

Each node on a SODA network consists of two processors: a **client processor**, and an associated **kernel processor**. The kernel processors are all alike. They are connected to the network and communicate with their client processors through shared memory and interrupts. Nodes are expected to be more numerous than processes, so client processors are not multi-programmed.

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

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

Processes are informed of interesting events by means of software interrupts. Each process establishes a single **handler** which it can close temporarily when it needs to mask out interrupts. A process feels a software interrupt when its id and one of its advertised names are specified in a request from some other process. The handler is provided with the id of the requester and the arguments of the request, including the out-of-band information. The interrupted process is free to save the information for future reference.

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

specifies buffer sizes. The amount of data transferred in each direction is the smaller of the specified amounts.

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

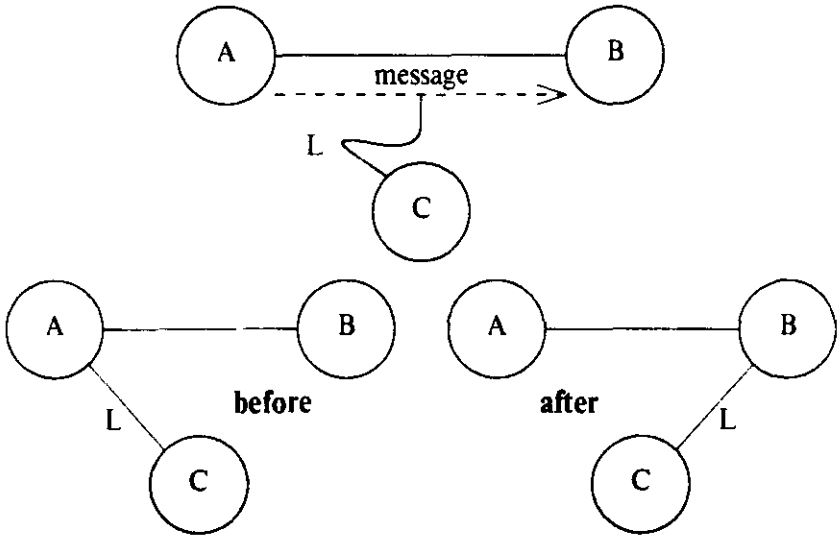
4.2. A Different Approach to Links

A link in SODA can be represented by a pair of unique names, one for each end. A process that owns an end of a link advertises the associated name. Every process knows the names of the link ends it owns. Every process keeps a **hint** as to the current location of the far end of each of its links. The hints can be wrong, but are expected to work most of the time.

A process that wants to send a LYNX message, either a request or a reply, initiates a SODA *put* to the process it thinks is on the other end of the link. A process moves link ends by enclosing their names in a message. When the message is SODA-accepted by the receiver, the ends are understood to have moved. Processes on the fixed ends of moved links will have incorrect hints.

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

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



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

the above example, C will re-start its request with B instead of A.

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

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

A process that is unable to find the far end of a link must assume it has been destroyed. If L exists, the heuristics of caching and broadcast should suffice to find it in the vast majority of cases. If the failure rate is comparable to that of other "acceptable" errors, such as garbled messages with "valid" checksums, then the heuristics may indeed be all we *ever* need.

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

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

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

This algorithm has the considerable disadvantage of bringing every LYNX process in existence to a temporary halt. On the other hand, it is simple, and should only be needed when a node crashes or a destroyed link goes unused for so long that everyone has forgotten about it.

4.2.1. Potential Problems As mentioned in the introduction, the SODA version of LYNX was designed on paper only. An actual implementation would need to address a number of potential problems. To begin with, SODA places a small, but unspecified, limit on the size of the out-of-band information for *request* and *accept*. If all the self-descriptive information included in messages under Charlotte were to be provided out-of-band, a minimum of about 48 bits would be needed. With fewer bits available, some information would have to be included in the messages themselves, as in Charlotte.

A second potential problem with SODA involves another unspecified constant: the permissible number of outstanding requests between a given pair of processes. The implementation described in the previous section would work easily if the limit were large enough to accommodate three requests for every link between the processes (a LYNX-request *put*, a LYNX-reply *put*, and a status *signal*). Since reply messages are always wanted (or can at least be discarded if unwanted), the implementation could make do with two outstanding requests per link and a single extra for

replies. Too small a limit on outstanding requests would leave the possibility of deadlock when many links connect the same pair of processes. In practice, a limit of a half a dozen or so is unlikely to be exceeded (it implies an improbable concentration of simultaneously-active resources in a single process), but there is no way to reflect the limit to the user in a semantically-meaningful way. Correctness would start to depend on global characteristics of the process-interconnection graph.

4.3. Predicted Measurements

Space requirements for run-time support under SODA would reflect the lack of special cases for handling unwanted messages and multiple enclosures. Given the amount of code devoted to such problems in the Charlotte implementation, it seems reasonable to expect a savings on the order of 4K bytes.

For simple messages, run-time routines under SODA would need to perform most of the same functions as their counterparts for Charlotte. Preliminary results with the Butterfly implementation (described in the following section) suggest that the lack of special cases might save some time in conditional branches and subroutine calls, but relatively major differences in run-time package overhead appear to be unlikely.

Overall performance, including kernel overhead, is harder to predict. Charlotte has a considerable hardware advantage: the only implementation of SODA ran on a collection of PDP-11/23's with a 1-Mbit/second CSMA bus. SODA, on the other hand, was designed with speed in mind. Experimental figures reveal that for small messages SODA was three times as fast as Charlotte.[2] Charlotte programmers made a deliberate decision to sacrifice efficiency in order to keep the project manageable. A SODA version of LYNX might well be *intrinsically* faster than a comparable version for Charlotte.

5. The Chrysalis Implementation

5.1. Overview of Chrysalis

The BBN Butterfly Parallel Processor [5] is a 68000-based shared-memory multiprocessor. The Chrysalis operating system provides primitives, many of them in microcode, for the management of system abstractions. Among these abstractions are **processes**, **memory objects**, **event blocks**, and **dual queues**.

Each process runs in an address space that can span as many as one or two hundred memory objects. Each memory object can be mapped into the address spaces of an arbitrary number of processes. Synchronization of access to shared memory is achieved through use of the event blocks and dual queues.

An event block is similar to a binary semaphore, except that 1) a 32-bit datum can be provided to the *V* operation, to be returned by a subsequent *P*, and 2) only the **owner** of an event block can **wait** for the event to be **posted**. Any process that knows the name of the event can perform the post operation. The most common use of event blocks is in conjunction with dual queues.

A dual queue is so named because of its ability to hold either data or event block names. A queue containing data is a simple bounded buffer, and enqueue and dequeue operations proceed as one would expect. Once a queue becomes empty, however, subsequent dequeue operations actually *enqueue* event block names, on which the calling processes can wait. An enqueue operation on a queue containing event block names actually posts a queued event instead of adding its datum to the queue.

---

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

5.2. A Third Approach to Links

In the Butterfly implementation of LYNX, every process allocates a single dual queue and event block through which to receive notifications of messages sent and received. A link is represented by a memory object, mapped into the address spaces of the two connected processes. The memory object contains buffer space for a single request and a single reply in each direction. It also contains a set of flag bits and the names of the dual 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 dual queue for the process at the other end of the link. When the process reaches a block point it attempts to dequeue a notice from its own dual queue, waiting if the queue is empty.

As in the SODA implementation, link movement relies on a system of hints. Both the dual queue names in link objects and the notices on the dual queues themselves are considered to be hints. Absolute information about which link ends belong to which processes is known only to the owners of the ends. Absolute information about the availability of messages in buffers is contained only in the link object flags. Whenever a process dequeues a notice from its dual 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 dual queue, but not every dual 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 dual queue, and *then* inspects the flags. It enqueues notices on its own dual queue for any of the flags that are set.

Primitives provided by Chrysalis make atomic changes to flags extremely inexpensive. Atomic changes to quantities larger than 16 bits (including dual queue names) are relatively costly. The recipient of a moved link therefore writes the name of its dual queue into the new memory object in a non-atomic fashion. It is possible that the process at the non-moving end of the link will read an invalid name, but only *after* setting flags. Since the recipient completes its update of the dual-queue name *before* inspecting the flags, changes are never overlooked.

Chrysalis keeps a reference count for each memory object. To destroy a link, the process at either end sets a flag bit in the link object, enqueues a notice on the dual queue for the process at the other end, unmaps the link object from its address space, and informs Chrysalis that the object can be deallocated when its reference count reaches zero. When the process at the far end dequeues the destruction notice from its dual queue, it confirms the notice by checking it against the appropriate flag and then unmaps the link object. At this point Chrysalis notices that the reference count has reached zero, and the object is reclaimed.

Before terminating, each process destroys all of its links. Chrysalis allows a process to catch all exceptional conditions that might cause premature termination, including memory protection faults, so even erroneous processes can clean up their links before going away. Processor failures are currently not detected.

5.3. Preliminary Measurements

The Chrysalis implementation of LYNX has only recently become available. It consists of approximately 3600 lines of C and 200 lines of assembler, compiling to 15 or 16K bytes of object code and data on the 68000. Both measures are appreciably smaller than the respective figures for the Charlotte implementation.

Message transmission times are also faster on the Butterfly, by more than an order of magnitude. Recent tests indicate that a simple remote operation requires about 2.4 ms with no data transfer and about 4.6 ms with 1000 bytes of parameters in both directions. Code tuning and protocol optimizations now under development are likely to improve both figures by 30 to 40%.

6. Discussion

Even though the Charlotte kernel provides a higher-level interface than does either SODA or Chrysalis, and even though the communication mechanisms of LYNX were patterned in large part on the primitives provided by Charlotte, the implementations of LYNX for the latter two systems are smaller, simpler, and faster. Some of the difference can be attributed to duplication of effort between the kernel and the language run-time package. Such duplication is the usual target of so-called **end-to-end arguments** [19]. Among other things, end-to-end arguments observe that each level of a layered software system can only eliminate errors that can be described in the context of the interface to the level above. Overall reliability must be ensured at the application level. Since end-to-end checks generally catch *all* errors, low-level checks are redundant. They are justified only if errors occur frequently enough to make early detection essential.

LYNX routines never pass Charlotte an invalid link end. They never specify an impossible buffer address or length. They never try to send on a moving end or enclose an end on itself. To a certain extent they provide their own top-level acknowledgments, in the form of goahead, retry, and forbid messages, and in the confirmation of operation names and types implied by a reply message. They would provide additional acknowledgments for the replies themselves if they were not so expensive. For the users of LYNX, Charlotte wastes time by checking these things itself.

Duplication alone, however, cannot account for the wide disparity in complexity and efficiency between the three LYNX implementations. Most of the differences appear to be due to the difficulty of adapting higher-level Charlotte primitives to the needs of an application for which they are almost, but not quite, correct. In comparison to Charlotte, the language run-time packages for SODA and Chrysalis can

(1) move more than one link in a message

(2) be sure that all received messages are wanted

(3) recover the enclosures in aborted messages

(4) detect all the exceptional conditions described in the language definition, without any extra acknowledgments.

These advantages obtain precisely because the facilities for managing virtual circuits and for screening incoming messages are *not* provided by the kernel. By moving these functions into the language run-time package, SODA and Chrysalis allow the implementation to be tuned specifically to LYNX. In addition, by maintaining the flexibility of the kernel interface they permit equally efficient implementations of a wide variety of other distributed languages, with entirely different needs.

It should be emphasized that Charlotte was not originally intended to support a distributed programming language. Like the designers of most similar systems, the Charlotte group expected applications to be written directly on top of the kernel. Without the benefits of a high-level language, most programmers probably would prefer the comparatively powerful facilities of Charlotte to the comparatively primitive facilities of SODA or Chrysalis. With a language, however, the level of abstraction of underlying software is no longer of concern to the average programmer.

For the consideration of designers of future languages and systems, we can cast our experience with LYNX in the form of the following three lessons:

Lesson one: Hints can be better than absolutes.

The maintenance of consistent, up-to-date, distributed information is often more trouble than it is worth. It can be considerably easier to rely on a system of hints, so long as they usually work, and so long as we can tell when they fail.

The Charlotte kernel admits that a link end has been moved only when all three parties agree. The protocol for obtaining such agreement was a major source of problems in the kernel, particularly in the presence of failures and simultaneously-moving ends [3]. The implementation of links *on top of* SODA and Chrysalis was comparatively easy. It is likely that the Charlotte

kernel itself would be simplified considerably by using hints when moving links.

Lesson two: Screening belongs in the application layer.

Every reliable protocol needs top-level acknowledgments. A distributed operating system can attempt to circumvent this rule by allowing a user program to describe *in advance* the sorts of messages it would be willing to acknowledge if they arrived. The kernel can then issue acknowledgments on the user's behalf. The shortcut only works if failures do not occur between the user and the kernel, and if the descriptive facilities in the kernel interface are sufficiently rich to specify precisely which messages are wanted. In LYNX, the termination of a coroutine that was waiting for a reply can be considered to be a "failure" between the user and the kernel. More important, the descriptive mechanisms of Charlotte are unable to distinguish between requests and replies on the same link.

SODA provides a very general mechanism for screening messages. Instead of asking the user to *describe* its screening function, SODA allows it to provide that function itself. In effect, it replaces a static description of desired messages with a formal subroutine that can be called when a message arrives. Chrysalis provides no messages at all, but its shared-memory operations can be used to build whatever style of screening is desired.

Lesson three: Simple primitives are best.

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

A high-level interface is only useful to those applications for which its abstractions are appropriate. An application that requires only a subset of the features provided by an underlying layer of software must generally pay for the whole set anyway. An application that requires features *hidden* by an underlying layer may be difficult or impossible to build. For general-purpose computing a distributed operating system must support a wide variety of languages and applications. In such an environment the kernel interface will need to be relatively primitive

Acknowledgments

Much of the research described in this article was conducted in the course of doctoral studies at the University of Wisconsin under the supervision of Associate Professor Raphael Finkel.

References

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

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

[3] Y. Artsy, H.-Y. Chang, and R. Finkel, "Charlotte: Design and Implementation of a Distributed Kernel," Computer Sciences Technical Report #554, University of Wisconsin – Madison, August 1984.

[4] Y. Artsy, H.-Y. Chang, and R. Finkel, "Interprocess Communication in Charlotte," Computer Sciences Technical Report #632, University of Wisconsin – Madison, February 1986. Revised version to appear in *IEEE Software*.

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

[6] A. P. Black, "Supporting Distributed Applications: Experience with Eden," *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, 1-4 December 1985, pp. 181-193. In *ACM Operating Systems Review 19*:5.

[7] N. Carriero and D. Gelernter, "The S/Net's Linda Kernel," *ACM TOCS 4*:2 (May 1986), pp. 110-129. Originially presented at the *Tenth ACM Symposium on Operating Systems Principles* Orcas Island, WA, 1-4 December 1985.

[8] D. R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, 10-13 October 1983, pp. 129-140. In *ACM Operating Systems Review 17*:5.

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

[10] R. Finkel, M. Solomon, D. DeWitt, and L. Landweber, "The Charlotte Distributed Operating System: Part IV of the First Report on the Crystal Project," Computer Sciences Technical Report #502, University of Wisconsin – Madison, October 1983.

[11] D. Gelernter, "Dynamic Global Name Spaces on Network Computers," *Proceedings of the 1984 International Conference on Parallel Processing*, 21-24 August 1984, pp. 25-31.

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

[13] J. Kepecs, "SODA: A Simplified Operating System for Distributed Applications," Ph. D. Thesis, University of Wisconsin – Madison, January 1984. Published as Computer Sciences Technical Report #527, by J. Kepecs and M. Solomon.

[14] J. Kepecs and M. Solomon, "SODA: A Simplified Operating System for Distributed Applications," *ACM Operating Systems Review 19*:4 (October 1985), pp. 45-56. Originally presented at the *Third ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing*, 27-29 August 1984.

[15] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM TOPLAS 5*:3 (July 1983), pp. 381-404.

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

[17] M. L. Powell and B. P. Miller, "Process Migration in DEMOS/MP," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, 10-13 October 1983, pp. 110-118. In *ACM Operating Systems Review 17*:5.

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

[19] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-To-End Arguments in System Design," *ACM TOCS 2*:4 (November 1984), pp. 277-288.

[20] M. L. Scott and R. A. Finkel, "LYNX: A Dynamic Distributed Programming Language," *Proceedings of the 1984 International Conference on Parallel Processing*, 21-24 August 1984, pp. 395-401.

[21] M. L. Scott, "Design and Implementation of a Distributed Systems Language," Ph. D. Thesis, Technical Report #596, University of Wisconsin – Madison, May 1985.

[22] M. L. Scott, "Language Support for Loosely-Coupled Distributed Programs," TR 183, Department of Computer Science, University of Rochester, January 1986. Revised version to appear in *IEEE Transactions on Software Engineering*, December 1986.

[23] R. E. Strom and S. Yemini, "NIL: An Integrated Language and System for Distributed Programming," *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, 27-29 June 1983, pp. 73-82. In *ACM SIGPLAN Notices 18*:6.

[24] R. E. Strom and S. Yemini, "The NIL Distributed Systems Programming Language: A Status Report," *ACM SIGPLAN Notices 20*:5 (May 1985), pp. 36-44.

[25] D. C. Swinehart, P. T. Zellweger, and R. B. Hagmann, "The Structure of Cedar," *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, 25-28 June 1985, pp. 230-244. In *ACM SIGPLAN Notices 20*:7 (July 1985).

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

[27] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, 10-13 October 1983, pp. 49-70. In *ACM Operating Systems Review 17*:5.

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