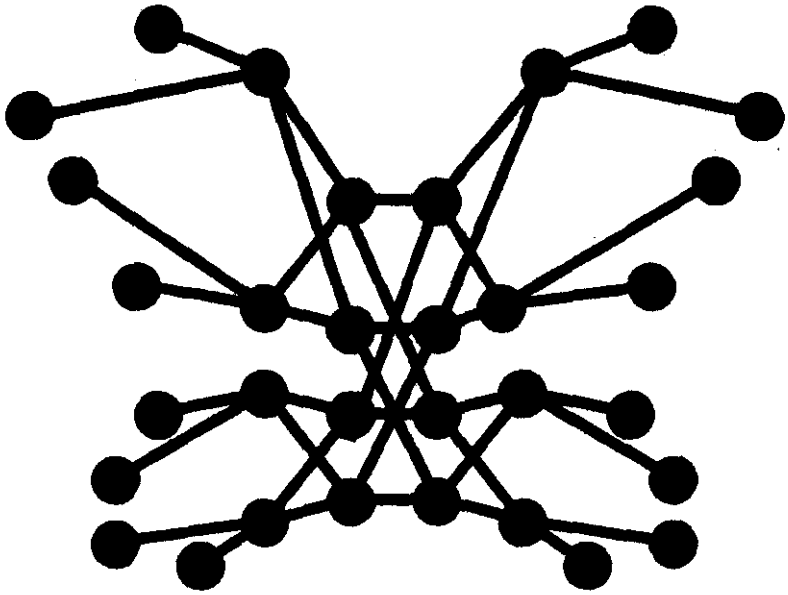

Butterfly Project Report
17

An Empirical Study of Message-Passing Overhead

Michael L. Scott and Alan L. Cox

December 1986

Computer Science Department
University of Rochester
Rochester, NY 14627

An Empirical Study of Message-Passing Overhead

Michael L. Scott
Alan L. Cox

University of Rochester
Computer Science Department
Rochester, NY 14627

December 1986

ABSTRACT

Conventional wisdom holds that message-passing is orders of magnitude more expensive than shared memory for communication between parallel processes. Differences in the speed of underlying hardware mechanisms fail to account for a substantial portion of the performance gap. The remainder is generally attributed to the "inevitable cost" of higher-level semantics, but a deeper understanding of the factors that contribute to message-passing overhead has not been forthcoming.

In this paper we provide a detailed performance analysis of one message-passing system: the implementation for the BBN Butterfly Parallel Processor of the LYNX distributed programming language. The case study includes a description of the implementation, an explanation of optimizations employed to improve its performance, and a detailed breakdown of remaining costs. The data provide a direct measure of the expense of individual features in LYNX. They also provide insight into the likely costs of other message-passing systems, both present and future. Lessons gained from our experience should be of use to other researchers in performing similar studies.

1. Introduction

On modern, microprocessor-based computers, the time required to load or store a word of main memory is on the order of microseconds, even in shared-memory multiprocessors. By contrast, for a very wide range of systems, the time required to pass a message between heavy-weight processes is on the order of tens of milliseconds, three or four orders of magnitude removed from memory access times.

There are at least two obvious reasons why messages should be more expensive than operations on shared memory. First, on many architectures there is a substantial fixed overhead associated simply with moving data from one place to another. For local area networks, this overhead is a function both of the bandwidth of the communication channel and of the processing time required to drive the interfaces. Cheriton and Zwaenepoel, for

This work was supported in part by NSF grant number DCR-8320136, by Darpa contract number DACA76-85-C-0001, and by an IBM Faculty Development Award.

example, report a **network penalty** of about a millisecond to send 100 bytes of data between SUN workstations on a 3 Mbit/second Ethernet [7].

The second explanation for the performance difference between shared memory and message passing is that messages provide semantics that shared variables do not. No message can be received before it is sent, so synchronization (and context switches) are inherent. In addition, most styles of message-passing provide queueing, flow control, and some form of authentication or protection. Many provide address resolution, type checking, exception handling, and gathering and scattering of parameters as well. On local area networks, most provide cleaner failure semantics than does the underlying hardware.

Even in the absence of physically-sharable memory, logically-shared variables can be implemented much more cheaply than can, for instance, remote procedure calls. With microcoded protocols, Spector [24] was able to perform remote memory operations in about 50 instruction times (155 μs) on Xerox Alto computers on a 3 Mbit/second Ethernet. The non-microcoded equivalent took 30 times as long, and was still four times faster than a request and reply message using standard protocols (see below). In his studies of remote operations in StarMod, LeBlanc [14] reported similar results: 150 instruction times (880 μs) to perform remote memory operations on PDP 11/23 computers connected by a 1 Mbit/second network. Remote procedure calls, by contrast, took over 20 times as long.

Despite the "obvious" reasons why message passing should take longer than reading and writing shared data, there seems to be a widespread impression among researchers in the field that messages take "too long". This impression is not new. The notes from a workshop on distributed computing, held at MIT in 1978 [17], contain the following statement:

> Message passing appears to be very expensive. Although it was difficult to establish uniform definitions of what was being measured, a time of about *20 milliseconds* was quoted as the round trip time to send a (null) message and receive an answer on the Xerox Alto systems, with similar numbers put forth for IBM systems and Multics. This is both a surprisingly high and surprisingly uniform time. . . . no one was able to indicate exactly where the time went, or why 20 ms should be a universal lower bound on message passing time.

Though processors, networks, and protocols have all improved dramatically since 1978, message-passing times are still relatively slow. Detailed figures are not always published, but users of Accent [19] and Mach [1] (with the Matchmaker stub generator [11]), Charlotte [3], Clouds [13], Demos/MP [18], and Eden [6] all report times in the tens of milliseconds to perform simple remote operations. The V kernel [8], which places great emphasis on speed, requires 1.46 ms for a request and reply within a machine, and 3.1 ms

between machines with SUN workstations on a 10 Mbit Ethernet. The award for fastest operations with the highest level of semantics probably goes to the RPCruntime package and Lupine stub generator, running in the Cedar environment at Xerox PARC [5]. Birrell and Nelson report a time for this system of approximately 1.1 ms per call to an empty remote procedure, with relatively high performance Dorado workstations connected by a 3 Mbit Ethernet.

In the quotation above, it is important to heed the caution about defining what is measured. Comparisons between disparate systems are limited not only by differences in the organization and speed of the underlying hardware, but also by differences in the level of semantics supported. In his doctoral thesis, Nelson reports a time of 145 *microseconds* per remote procedure call in his fastest Dorado implementation. The time drops to 124 μs if processes busy-wait. This implementation is so fast that the 11 μs overhead of the timing loop becomes a significant fraction of the total cost. Unfortunately, the semantics provided are so restricted that the timings do little more than provide a lower bound for the cost of RPC: parameters are limited to a few basic types, manually gathered and scattered; stubs must be coded manually as well; the communication firmware is implemented in such a way that standard Ethernet protocols are unavailable to other processes on the machine.

Rather than fuel a debate over whose software is fastest, we prefer to ask for each individual system: what are the factors that contribute to its overhead? Careful attribution of costs to message-passing subtasks is crucial for optimization. Moreover, it is the only reliable way to evaluate the cost-effectiveness of contemplated features. Without detailed accounting, it is impossible to determine whether the difference in speed between competing systems is due to hardware overhead, choice of semantics, or simply cleverness of coding.

Surprisingly, very little such accounting has made its way into print. Spector is almost alone in providing a microsecond-by-microsecond breakdown of the time required to perform a remote operation. It is our intention in this paper to provide an equally detailed analysis for remote procedure calls in a distributed programming language. We believe this analysis to be useful not only in the understanding of our own particular language and implementation, but also in the design of similar systems and in the development of an intuitively satisfying appreciation of "where the time goes" in message-passing systems in general.

Our choice of language is LYNX [23], as implemented [22] on the BBN Butterfly Parallel Processor [4]. We believe LYNX to be representative of a large class of languages in which interprocess communication is based on rendezvous or remote procedure call.

Languages in this class include Ada [26], Argus [15], NIL [25], SR [2], and the dialects of CSP [10]. The Butterfly, with its shared-memory architecture, is in some ways quite unlike the more common message-based multicomputers, but the difference between multiprocessor block transfers and multicomputer messages has a relatively small and self-contained impact on the protocols required to implement remote procedure calls. Moreover, it is the nature of performance studies such as these that the peculiarities of a particular language and machine are readily identified and isolated in the final data.

Section 2 of this paper provides an overview of LYNX. It is followed by a description of our implementation for the Butterfly (section 3) and of the optimizations we applied to improve its performance (section 4). Section 5 contains an accounting, by subtask, of the costs that remain. We conclude with a discussion of the meaning of those costs.

2. Overview of LYNX

The LYNX programming language is not itself the subject of this article. Language features and their rationale are described in detail elsewhere [21,23]. Our intent in the current section is to provide only as much information as is needed to understand the remainder of the paper.

The fundamental abstractions in LYNX are the **process**, the **link**, and the **thread of control**. Processes execute in parallel, possibly on separate processors. There is no provision for shared memory. Processes interact only by exchanging messages on links. A link is a two-directional communication channel with a single process at each end. Each process may be divided into an arbitrary number of threads of control. Threads are similar to coroutines; they are a control-flow mechanism that facilitates the management of multiple contexts within a process. Threads are not designed for parallelism; they execute in mutual exclusion. Execution moves to another thread automatically when the current thread is blocked.

New threads may be created at any level of lexical nesting. Threads that are nested inside the same scope may therefore share non-local, non-global data. The activation records of a process form a tree (a **cactus stack**), with a thread of control in each leaf. Looking back up the path to the root, each thread sees what appears to be a normal run-time stack. Individual activation records are allocated dynamically in a number of standard sizes, much as they are in Mesa [12].

Interprocess communication is based on the invocation of remote operations. A process that wishes to provide a given operation can do so in one of two ways: it can create a thread of control that waits for a request explicitly, or it can **bind** a link to an entry

procedure so that a new thread will be created automatically to handle each incoming request. The explicit case is similar to the **accept** statement of Ada, and shares its name. The implicit case is an example of a remote procedure call.

A thread requests a remote operation by executing a **connect** statement. It blocks until a reply is received. Meanwhile, other threads may execute. Remote operations are therefore non-blocking from the point of view of a *process*.

Incoming messages 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 all the threads in the process are blocked (at which time we say the process has reached a **block point**). A reply queue is open whenever a thread has sent a request on the link and has yet to receive a reply. A request queue is open whenever its link has been bound to an entry procedure or named by a thread that is waiting to *accept*.

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 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 thread within the sending process. The process must be notified when messages are received in order to unblock appropriate threads. 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 threads in the run-time package for each sending process.

One of the more challenging features 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 a moved link must be oblivious to the move, even if it is currently relocating its end as well.

3. Initial Implementation

3.1. The Butterfly and Chrysalis

The BBN Butterfly Parallel Processor [4] can support up to 256 separate processing nodes. Each node consists of a Motorola 68000 CPU, a bit-sliced co-processor called the **Processor Node Controller**, (PNC) and up to 4 Mbytes of RAM. The 68000 runs at 8 MHz. An empty subroutine call with no parameters (JSR, LINK, ULNK, RTS) completes in almost exactly 10 μs. Newer machines employ a 68020 and 68881, with a double-speed clock.

The PNCs are connected by the Butterfly Switch, an FFT-style interconnection network. Each PNC mediates all memory requests from its processor, passing them through to local memory when appropriate, or forwarding them through the switch to another PNC. References to individual words of remote memory take 3 to 5 times as long as references to local memory. The PNCs also provide atomic fetch-and-phi operations, as well as a microcoded block transfer that achieves an effective throughput between nodes of about 20 Mbits/sec, with a start-up cost of 50 μs.

The Butterfly's native operating system, called Chrysalis, provides primitives for the management of a number of basic abstractions, including **processes**, **memory objects**, **event blocks**, and **dual queues**. Many of the primitives are supported by PNC microcode.

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 will post a queued event instead of adding its datum to the queue.

3.2. LYNX Compiler and Run-time System

LYNX is implemented by a cross compiler that runs on the Butterfly's host machine. 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.

Communication between LYNX processes is supported by a run-time library package, also written in C. At start-up time, every LYNX 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 (see figure 1). Within each process, the link is referenced by indexing into an array of link descriptors in the run-time support package. Each descriptor contains a pointer to the shared memory object, together with lists of threads that are waiting for communication on the link. The memory object itself 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.

In addition to message buffers, each link object 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

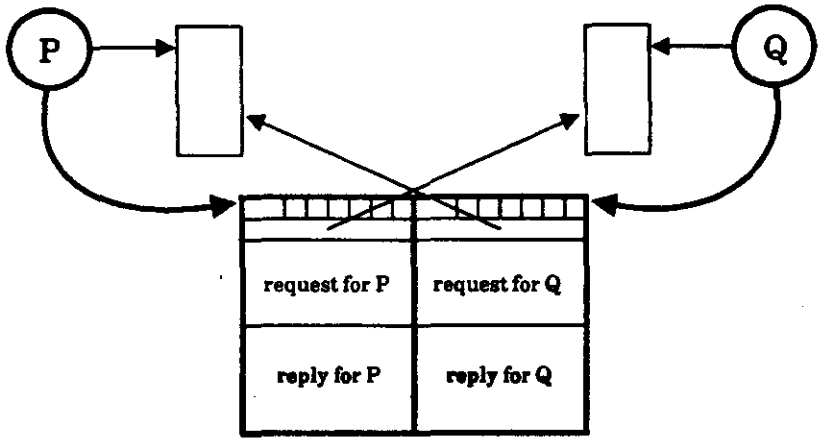

Figure 1: shared link object

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.

The flag bits permit the implementation of link movement. 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.

3.3. Protocol

Notifications on dual queues, with block transfers for data movement, play the role of messages in our implementation. Our initial protocol defined eight types of notices: REQ, REQ_ACK, REP, REP_ACK, FAR_END_DESTROYED, REQ_ACK_ERR, REP_ERR, and REP_ACK_ERR. The final three are used only in the event of exceptions, type clashes, or requests for nonexistent operations. FAR_END_DESTROYED is used only when cleaning up connections. The rest of this discussion focuses on REQ, REQ_ACK, REP, and REP_ACK.

Suppose processes P and Q are connected by link L, and that a thread A in P wishes to invoke an operation provided by a thread B in Q. A blocks until the request buffer for Q is available in L. It fills the buffer, sets the appropriate flag, and posts a REQ notice on Q's dual queue. The next time Q is blocked, it receives the REQ notice and wakes up B. B copies the request out of the buffer into local variables, sets the appropriate flag, and posts a REQ_ACK notice on P's dual queue. When it receives this notice, P knows that the request buffer is available to threads other than A, if needed. When B is done serving the request, it blocks until the reply buffer for P is available in L. It fills the buffer, sets the appropriate flag, and posts a REP notice on P's dual queue. The next time P is blocked, it receives

the REP notice and wakes up A. A copies the reply out of the buffer into local variables, sets the appropriate flag, and posts a REP_ACK notice on Q's dual queue. When it receives this notice, Q unblocks B.

Since "messages" on the Butterfly are as reliable as main memory, acknowledgments are not needed to recover from "lost packets". They are required, however, for flow control and for confirmation of high-level semantic checks. In the event that P has no additional threads waiting to send requests to Q, the REQ_ACK notice can be eliminated (though the corresponding flag cannot). With relatively minor changes to the semantics of LYNX, the REP_ACK notice can be eliminated also. We explore these possibilities (among others) in section 4.

4. Optimizations

When our first timing figures were collected, we had not yet completed the code to establish links between independent processes. We were able, however, to create a link whose ends were both owned by the same process. We arranged for that process to send messages to itself. The "round trip" time for an null invocation came to 5.9 milliseconds. Through a series of four revisions, this time was reduced to 2.78 ms:

(1) Instruction histograms (from an execution profiler) indicated that the section of code consuming the largest individual amount of time was the standard integer multiplication subroutine (the 68000 does not have a 32-bit multiply instruction). Investigation revealed that the only reason the subroutine was being called was to calculate subscripts into the array of link descriptors in the run-time support package. Since each descriptor was 60 bytes long, the addition of a 4-byte pad allowed the generation of left shifts for multiplication. Total savings: 22%.

(2) Turning on peephole optimization in the C compiler and using conditional compilation to disable debugging support reduced the time to 4.0 ms/invocation. Additional savings: 13%.

(3) The original implementation of the cactus stack used the standard C **malloc** library to allocate activation records. We expected this to be slow, and profiling confirmed the expectation. The new allocator keeps a cache of frames in a number of "standard" sizes. Additional savings: 25%.

(4) In August of 1986 we took delivery of a new C compiler for the Butterfly, obtained by BBN from Green Hills Software. Use of the new compiler as the LYNX back end

resulted in better code. Additional savings: 7%.[1]

With the implementation complete and with obvious inefficiencies removed, we proceeded to a series of multi-process timing tests. Statistics were collected for simple programs that perform a large number of remote operations in a tight loop. Dividing total elapsed time by the number of iterations yields an estimate of the overhead of an individual operation.

This technique has several limitations. It ignores the effects of the scheduling algorithm, which may be atypical for a process that blocks frequently. It ignores the fact that a typical LYNX process is composed of a large number of threads and that several operations may be pending at once. It ignores the fact that each processor will usually be shared by a large number of processes, so that the latency seen by a single process may not reflect overall throughput for the jobs on the machine.

Despite its limitations, we have concentrated on round-trip latency because it is relatively easy to measure (a "representative" job mix is not required) and because it has been used to describe the performance of a large number of similar systems. Our code performs four basic tests: two for a null operation with no parameters, and two with 1000 bytes of parameters in each direction. In each case, one test uses implicit receipt (RPC) and the other explicit receipt (**accept**). After the arrival of the Green Hills C compiler, our figures for explicit receipt were as follows:[2]

	Processes on different nodes	Processes on same node
nullop	2.16 ms	3.55 ms
bigop	3.79 ms	5.19 ms

In considering the details of the protocol, we came to the realization that in many (if not most) cases, the REQ_ACK notice serves no purpose. It can be subsumed in the REP notice whenever the client process is not in a hurry to reuse the link's request buffer.[3] A new version of the run-time package was constructed that contains two additional flags in the shared link object. The flags indicate whether the two processes at the ends of the link are interested in receiving REQ_ACK notices. A client process sets the bit when it has

[1] The difference between the compilers is more pronounced in realistic programs. Our tests use relatively simple straight-line code, with very little in the way of complicated expressions or loops.

[2] Results are accurate to about ± 0.02 ms.

[3] LYNX links are completely symmetric. Either of the processes attached to a link can make requests of the other. We use the terms "client process" and "server process" to mean "the process playing the role of client (server, respectively) in the current discussion".

additional threads waiting to send requests. A server process posts a REQ_ACK notice on its partner's dual queue only when the 'interested' bit is set. The REQ_ACK bit is still set in any case. If a thread in the client tries to send a request while the request buffer appears to be full, the client will check the REQ_ACK bit to see if a potentially useful notice went unposted, and will re-create it if necessary.

With a minor change in the semantics of LYNX, the REP_ACK notice can be eliminated also. Like the REQ_ACK notice, REP_ACK serves to inform a process that a buffer has been emptied and can be used by another thread of control. In addition, it serves to inform a server that the requesting thread in the client was still waiting when the reply arrived. LYNX semantics call for the server thread to feel an exception if the client thread has died (as a result of feeling an exception itself). For efficiency reasons, the original implementation of LYNX (on the Crystal multicomputer at the University of Wisconsin [9]) did not support these semantics, and it would not be a serious loss to forgo them on the Butterfly as well. We constructed a version of the run-time package in which another two flags were added to the link object, much like the 'interested' bits above. REP_ACK notices are eliminated when the server process has no additional threads waiting to send replies. Adoption of the modification was deferred until the size of the potential time savings could be determined.

Figure 2 compares the three versions of the protocol under the assumption that the optional notices are never required. Again with explicit receipt, our times for the modified protocols are as follows:

No request acknowledgments:

	Processes on different nodes	Processes on same node
nullop	2.22 ms	3.26 ms
bigop	3.67 ms	4.90 ms

No request or reply acknowledgments:

	Processes on different nodes	Processes on same node
nullop	1.96 ms	2.82 ms
bigop	3.59 ms	4.42 ms

Until we collected timing results and constructed figure 2, we did not realize that with the client and server on separate nodes the principal effect of the protocol changes would be to reduce the amount of overlapped computation, without reducing latency. The savings for processes on the same node were more than twice as large, percentage-wise, as the savings for processes on separate nodes. For the null operation, latency on separate nodes actually *increased* slightly when request acknowledgments were removed (though it dropped below the original figure when reply acknowledgments were removed as well). A full explanation of the figures depends on at least three factors:

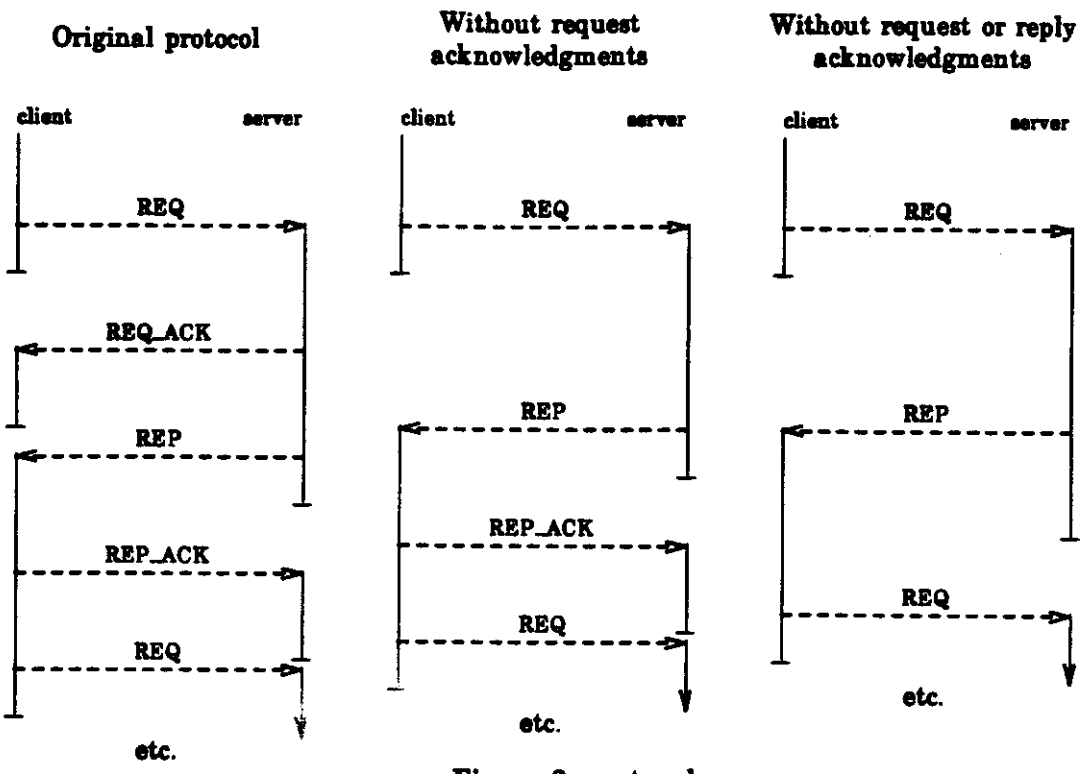

Figure 2: protocols

(1) In the first and second protocols, the timing loops for nullop are tight enough that the processes do not have quite enough time to finish examining an acknowledgment before the next notice arrives. The requests and replies that follow acknowledgments are therefore received without waiting. For bigop, the extra time required to copy parameters means that the processes wait for every notice.

(2) It takes less time to post a notice to an empty dual queue than it does to post to a queue on which another process is waiting. In the first protocol, the server posts two notices: the REQ_ACK and the REP. For bigop, both of the posts are expensive. For nullop, only one of them is. The second protocol therefore saves more time in the server for bigop than it does for nullop.

(3) The client has work it must do when it knows that its request has been received. The second protocol eliminates the overhead of one invocation of the dispatcher, but some of the work that used to be overlapped with the server when the REQ_ACK was received must now be done while the server is blocked, after the REP is received.

In moving from the first to the second protocol, non-overlapped time is saved in the client when sending the REQ_ACK. Non-overlapped time is lost in the client after receiving the REP. There is a net gain for bigop. There is a net loss for nullop, because of the tight loop and lack of waiting. In moving from the second to the third protocol, the semantic changes to LYNX allow most of the work that was performed upon receipt of a REP_ACK to be eliminated, not deferred (the thread of control that sends a reply no longer blocks). There is therefore a net gain for both bigop and nullop.

5. Remaining Costs

For more detailed protocol analysis, our principal tool was an execution-time profiler that builds histograms from periodic samplings of the program counter. Our C compiler does not support the collection of subroutine call counts, but the protocol is simple enough for them to be predicted by hand. Some of our early optimizations, particularly the change in size of the link record and the replacement of the activation record allocator, were motivated by profiling results. Those results were examined, however, at the granularity of procedure calls only. In the analysis reported in this section, we worked at the level of individual instructions.

Statistics were collected for the selfloop program described at the beginning of section 4. The program was run for 100,000 iterations, with a client thread making requests of a server thread over a link that connected to the same process at both ends. Assembly listings of the run-time support package were compared against the C-language source to determine the purpose of each individual instruction. The counts for instructions with similar purposes were grouped together into categories. The results are summarized in figure 3.

5.1. Explanation

Threads

The run-time package maintains a ready list, together with lists of threads waiting for various kinds of messages. There is overhead associated with moving threads between lists, with saving and restoring context at thread switches, with verifying that buffers are available when a thread wishes to communicate, and with searching lists for appropriate threads when notices arrive. The ability to create nested threads leads to dynamic allocation of activation records.

Checking and exception handling

Links must be checked for validity at every **connect** and **accept** statement. Since

Threads		
queue management	4.8	
context switches	6.4	
buffer acquisition	1.3	
queue searching (dispatcher)	3.1	
cactus stack	6.8	22.4
Checking and exception handling		
are connect and accept links valid?	1.6	
is notice link valid (dispatcher)?	1.6	
run-time type checking (dispatcher)	0.8	
general overhead for LYNX exceptions	0.6	
establishment of LYNX exception handlers	4.4	
establishment of Chrysalis exception handler	9.2	18.1
Miscellaneous overhead		
client and server for loops	0.6	
dispatcher while loop, switch	1.4	
procedure-call overhead for communication routines	14.7	
loading of registers with active values	5.6	22.2
Bookkeeping (who wants what sorts of services and who is willing to provide them)		10.6
Actual communication		
set, clear flags	12.4	
enqueue, dequeue notices	7.3	
find addresses of buffers	1.0	20.6
Protocol option testing		
link movement	2.0	
background threads	1.0	
premature requests	1.6	
optional acknowledgments	1.8	6.4

Figure 3: cost breakdown (in percent of total work performed)

dual queue notices are hints, the link mentioned in an incoming notice must be checked for validity as well. LYNX relies on run-time type checking for messages, but the overhead is very low [20]. Much larger amounts of time are devoted to setting up and taking down exception handlers.

LYNX provides an exception-handling mechanism similar to that of Ada [26]. The implementation requires a single, 32-bit move instruction at the beginning of each subroutine, and a somewhat larger amount of work at the beginning and end of each handler-protected block of code. Errors in communication result in exceptions in appropriate threads. Modularity of the run-time package is maintained by enclosing parts of the protocol in default exception handlers that put their data structures into a

consistent state and then re-raise the exception.

The Chrysalis operating system itself provides another form of exception handling, grafted onto the C language through use of the C preprocessor. Instead of returning a failure code, an operating system service that is unable to complete successfully will cause a Chrysalis **throw**. Since they are not supported directly by the compiler, the **catch blocks** that handle throws impose a larger cost than do the handlers of LYNX. There is only one catch block in the language run-time package. It protects the enqueue operation when posting notices, and is therefore set up and taken down twice per iteration, consuming over nine percent of the total execution time.

Miscellaneous overhead

The **for** loops in the client and server are self-explanatory, as are the calling sequences for subroutines. The loop in the dispatcher keeps dequeueing notices until one of them can be used to make some thread runnable. The **switch** (**case**) statement has arms for each kind of incoming message.

The C compiler is clever enough to move frequently-used values into registers at the beginning of each subroutine. It is essentially impossible to attribute the cost of doing so to individual instruction categories.

Bookkeeping

When a client thread requests an operation, the name of the operation, an encoding of the types of its parameters, and the name of the thread itself must all be written into the shared link object. When a server attempts to *accept* a message, similar information must be placed into data structures accessible to the dispatcher. Active servers must keep track of (possibly nested) clients waiting for replies. Link numbers and notice types must be packed and unpacked in notices.

Communication

Actual communication involves setting and clearing flags, enqueueing and dequeueing notices, and copying parameters. In our nullop tests, the third item consists simply of moving the addresses of buffers into pointers that are never used. In the absence of acknowledgment notices, there are four pairs of flag operations and two pairs of dual queue operations. The dual queue operations are more expensive individually, but less expensive collectively.

Protocol option testing

There are six places in the protocol at which special action must be taken to deal with moving link ends. At the top of the dispatcher's main loop there is a check that

returns control to a "background" thread if the notice queue is empty.[4] At the beginning of the code for **accept**, there is a check to see if a request notice was received before any thread was ready to provide the appropriate service. At the beginning of the code to post notices, there is a check that skips the enqueue of acknowledgments. None of the protocol's special cases arise in our simple timing tests, but the **if** statements that check for them account for over six percent of the total work performed.

5.2. Timeline

The instruction histogram counts from our timing tests can be used to build a timeline for remote invocations. The dimensions of figure 4 are based on the timings of the selfloop program, but are charted to indicate the operation of the nullop test with client and server processes on separate nodes. The time required to wake up a process has been estimated by subtracting the time for the selfloop test from the time for the nullop test (with client and server on the same node) and dividing by two. The length of the resulting timeline is 2.00 ms, a value that varies from the actual measured time for nullop on separate processors by just over two percent.

6. Discussion

6.1. Marginal Costs

6.1.1. Threads

Support for multiple threads of control within a process consumes over 22 percent of the total CPU time for a remote invocation. Management of the cactus stack is the largest single contributor to this total, but context switches and queue management run a close second and third.

Some of the overhead of threads could be reduced by changes to the implementation. Queue management might be cheaper on machines (such as the VAX) with hardware queue instructions. Alternatively, the time spent moving threads between queues could be eliminated by keeping all threads on a single linked list and performing a linear search of that list whenever a thread of a certain class was desired. Such a change would improve the timing results for our simple test programs, but would impose serious costs on practical

[4] There is no provision for asynchronous receipt of messages in LYNX, but a thread that has a large amount of low-priority work to do can poll for messages by indicating its desire to wait until communication has subsided.

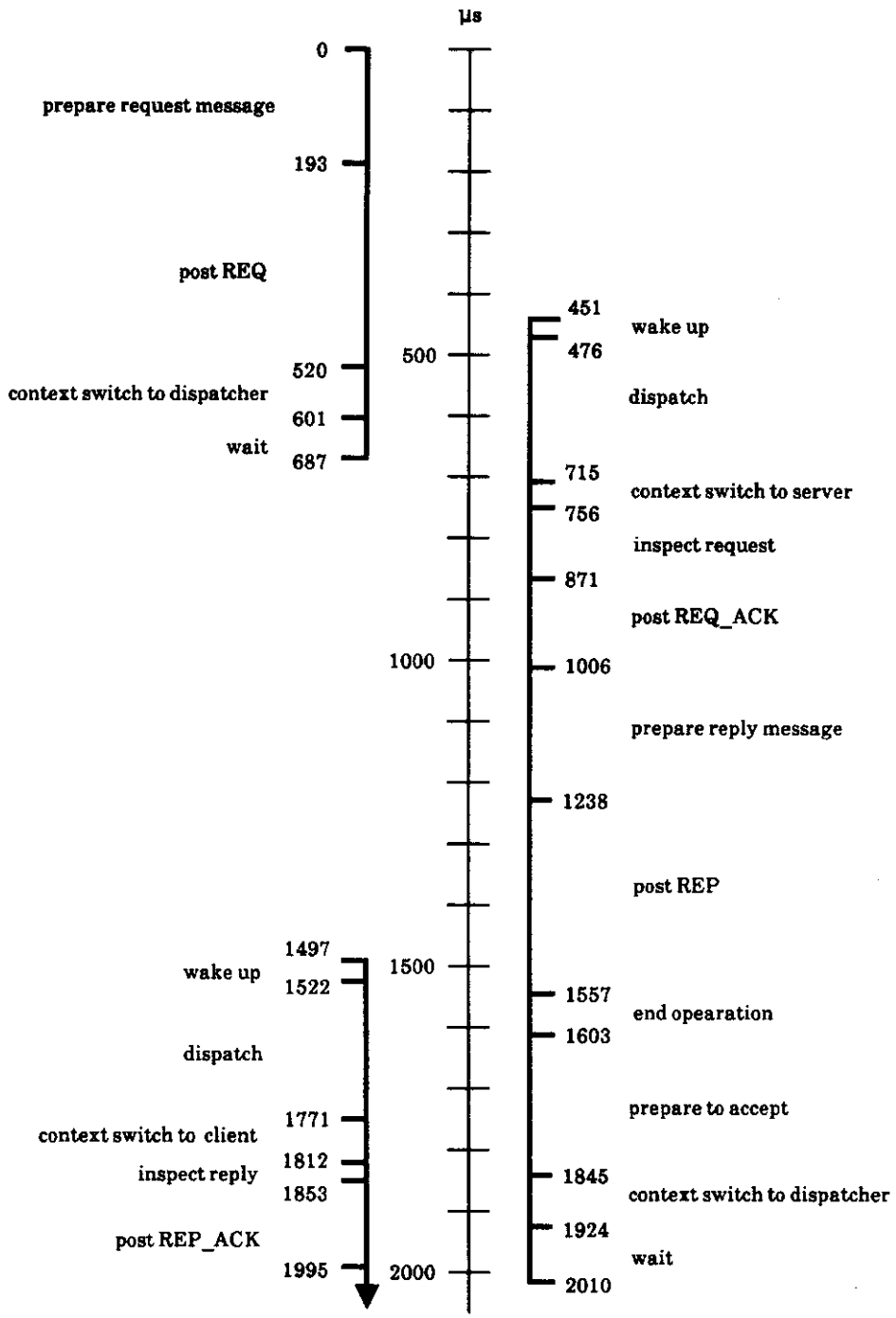

Figure 4: timeline

programs with very large numbers of threads.

The overhead of the cactus stack could be cut in half by a production-quality LYNX compiler. Because the current implementation uses the C compiler as a back end, it is not possible to determine the sizes of activation records until program start-up time. Instead of being hard-coded with a known frame size, the allocator is implemented as a parameterized macro; it pays for indexing operations at every subroutine call, in order to find the head node for an appropriate list of frames.

The cactus stack could be eliminated entirely by requiring all threads to be created at the outermost lexical level. Such a restriction would be consistent with the designs of several other distributed languages. For reasons explained elsewhere [23], we believe the ability to nest threads to be an important advantage of LYNX over other language designs, and would be reluctant to forgo it. We consider the measured overhead to be an acceptable price to pay.

Most of the functionality of threads, we believe, will be required in any programming system designed to support server process and a remote-invocation style of interprocess communication. This conclusion is supported by the work of Liskov, Herlihy, and Gilbert [16], and by the designers of a large number of other distributed programming languages, in which lightweight processes (usually designed to run in a truly parallel fashion) can be created in each address space.

There are good arguments both for and against the parallel execution of threads. It is likely, however, that any implementation supporting simultaneous execution of lightweight processes will be more expensive than the coroutine threads of LYNX. Our figures thus approximate a lower bound on queue manipulation and context-switching overhead.

6.1.2. Movable Links

Like the ability to nest threads of control, the movement of link ends is an important and distinctive characteristic of LYNX. We had hoped in our implementation to pay for moving ends only in the threads and processes that use them. It is of course necessary in each message to check whether link ends are enclosed. Figure 3 shows that those checks accounted for just over two percent of the elapsed time in our selfloop test. What is less obvious is that much of the time listed under "actual communication" can be attributed to moving links as well.

In the original protocol, with acknowledgments, the flag bits in the shared link objects were needed *only* to ensure the ability to move links. If it had not been possible to change the process at the far end of a link, then dual-queue notices could have provided absolute information instead of hints. With the development of the second and third

protocols of figure 2, the flag bits took on a second role: keeping track of acknowledgments that were not sent, but might have been if they had been wanted. Without those bits, a thread that attempted to send a request or reply would not know if the required buffer was still full, or had been emptied the instant before.

Without the need to move links, the newest protocol would be able to avoid setting and clearing bits for REQ and REP notices, but would still need to manipulate them for REQ_ACK and REP_ACK. Between the checks for enclosures and the setting and clearing of half of the flags, the marginal cost of the movability of links appears to be about nine percent of latency, or 180 μs per remote invocation.

We have considered an alternative implementation of link movement, in which dual queue notices carry absolute information. The overhead of flag bits would be eliminated for ordinary messages, but the cost of actually moving a link would increase dramatically. Link movement is important and frequent enough to make the tradeoff unattractive.

6.2. Lessons

For the benefit of those who may wish to undertake similar performance studies for other message-passing systems, we offer the following suggestions:

Intuition is not very helpful.

Beginning programmers are taught to distrust their intuition when attempting to tune their code. Our experience testifies to the wisdom of this advice. It came as a complete surprise, for example, when we discovered that we were spending 1.3 ms per invocation calculating subscripts into the table of link records. It was also a surprise (though a less happy one) when the elimination of acknowledgment notices on the dual queues yielded only modest improvements in latency. Similarly, we were disappointed to discover that the flag operations that permit link movement were responsible for as much as seven percent of our invocation time. We had been inclined to think of those operations as trivial.

Overlapped computation is crucial.

As demonstrated by our experience with the protocol optimizations of section 4, no explanation of message-passing overhead can be complete without an understanding of precisely which parts of the protocol can be executed simultaneously on separate processors.

It helps to have a variety of measurement techniques.

Our analysis drew on several kinds of statistics: we collected instruction count histograms; we timed communication within a process, between processes, and between

machines; we collected statistics (in the run-time support package) on such things as the number of times a process was forced to wait for a notice from its dual queue. No one of these measurement techniques alone sufficed to explain performance. The comparison of nullop timings with the predicted results of the timeline in section 5.2 provided a reassuring cross-check on our figures. The slowdown of the nullop test on separate machines after the elimination of REQ_ACK notices was explained by counting the number of times each process was blocked by dequeue operations.

7. Recent Results

Since completing the analysis of section 5, we have implemented several minor changes to the run-time package for LYNX. One of these circumvents the normal mechanism for establishing Chrysalis exception handlers, saving about 40 μs per enqueue operation when posting notices. Others changes result in small savings throughout the code, with relatively little impact on the *proportions* of figure 4. The following is a complete set of timings as of December 1986:

Explicit receipt:

	Processes on different nodes	Processes on same node
nullop	1.80 ms	2.58 ms
bigop	3.45 ms	4.21 ms

Implicit receipt:

	Processes on different nodes	Processes on same node
nullop	2.04 ms	2.76 ms
bigop	3.72 ms	4.42 ms

The figures for implicit receipt are larger than those for explicit receipt because of the overhead of creating and destroying threads. Space considerations for this paper preclude a detailed discussion.

Acknowledgment

Thanks to Ken Yap for his help in porting LYNX to the Butterfly.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986.

[2] G. R. Andrews, R. A. Olsson, M. Coffin, I. J. P. Elshoff, K. Nilsen, and T. Purdin, "An Overview of the SR Language and Implementation," TR 86-6a, Department of Computer Science, University of Arizona, 18 February 1986, revised 23 June 1986. Submitted to *ACM TOPLAS*.

[3] 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*.

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

[5] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM TOCS 2*:1 (February 1984), pp. 39-59. Originally presented at the *Ninth ACM Symposium on Operating Systems Principles*, 10-13 October 1983.

[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] 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.

[8] D. R. Cheriton and W. Zwaenepoel, "Distributed Process Groups in the V Kernel," *ACM TOCS 3*:2 (May 1985), pp. 77-107.

[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] C. A. R. Hoare, "Communicating Sequential Processes," *CACM 21*:8 (August 1978), pp. 666-677.

[11] 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.

[12] B. W. Lampson and D. D. Redell, "Experience with Processes and Monitors in Mesa," *CACM 23*:2 (February 1980), pp. 105-117.

[13] R. J. LeBlanc and C. T. Wilkes, "Systems Programming with Objects and Actions," *Proceedings of the Fifth International Conference on Distributed Computing Systems*, 13-17 May 1985, pp. 132-139.

[14] T. J. LeBlanc and R. P. Cook, "An Analysis of Language Models for High-Performance Communication in Local-Area Networks," *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, 27-29 June 1983, pp. 65-72. In *ACM SIGPLAN Notices 18*:6.

[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] B. Liskov, M. Herlihy, and L. Gilbert, "Limitations of Remote Procedure Call and Static Process Structure for Distributed Computing," Programming Methodology Group Memo 41, Laboratory for Computer Science, MIT, September 1984, revised October 1985.

[17] J. L. Peterson, "Notes on a Workshop on Distributed Computing," *ACM Operating Systems Review 13*:3 (July 1979), pp. 18-27.

[18] 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.

[19] 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.

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

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

[22] M. L. Scott, "The Interface Between Distributed Operating System and High-Level Programming Language," *Proceedings of the 1986 International Conference on Parallel Processing*, 19-22 August 1986, pp. 242-249. Also published in the *University of Rochester 1986-87 Computer Science and Computer Engineering Research Review*, and available as TR182 and BPR 6, Department of Computer Science, University of Rochester, September 1986 (revised).

[23] M. L. Scott, "Language Support for Loosely-Coupled Distributed Programs," *IEEE Transactions on Software Engineering*, January 1987. Also published as TR183, Department of Computer Science, University of Rochester, September 1986 (revised).

[24] A. Z. Spector, "Performing Remote Operations Efficiently on a Local Computer Network," *CACM 25*:4 (April 1982), pp. 246-260.

[25] 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.

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