Experience with Charlotte: Simplicity versus Function In a Distributed Operating System

by

Raphael A. Finkel
Michael L. Scott
William K. Kalsow
Yeshayhu Artsy, Hung-Yang Chang, Prasun Dewan, Aaron J. Gordon
Bryan Rosenburg, Marvin H. Solomon, Cui-Qing Yang

Computer Sciences Technical Report #653
July 1986

Experience with Charlotte: Simplicity versus Function in a Distributed Operating System

Raphael A. Finkel
Michael L. Scott
William K. Kalsow

Yeshayahu Artsy, Hung-Yang Chang, Prasun Dewan, Aaron J. Gordon,
Bryan Rosenburg, Marvin H. Solomon, Cui-Qing Yang

University of Wisconsin — Madison
1210 W. Dayton Street
Madison, WI 53706

This paper presents a retrospective view of the Charlotte distributed operating system, which is intended as a testbed for developing techniques and tools for exploiting large-grain parallelism to solve computation-intensive problems. Charlotte was constructed over the course of approximately 5 years, going through several distinct versions as the underlying hardware and our ideas for implementation changed. Charlotte rests on several underlying design decisions: (1) it is a software layer on the Crystal multicomputer, (2) processes do not share memory, (3) communication is on reliable, symmetric, bi-directional paths named by capabilities, and (4) absolute information is stored at each end of a communication path. Our implementation taught us that our dual goals of simplicity and function were not easily reached. In particular, the issue of simplicity is quite complex; quests for simplicity in various areas often conflict with each other. This paper explores how the design decisions we made to satisfy our goals incurred implementation cost and required extra levels of software, but resulted in a high-quality testbed for experimentation in distributed algorithm design.

Experience with Charlotte: Simplicity versus Function in a Distributed Operating System*

1. Introduction

Charlotte is a distributed operating system currently in production use at the University of Wisconsin–Madison [Artsy84, Artsy86]. Charlotte is intended as a testbed for developing techniques and tools for exploiting large-grain parallelism to solve computation-intensive problems. It runs on the Crystal network [DeWitt84], which contains 20 VAX-11/750 computers interconnected by an 80 Mbps token ring. Charlotte was constructed over the course of approximately 5 years, going through several distinct versions as the underlying hardware and our ideas for implementation changed. This paper presents a retrospective view of the Charlotte project.

Our starting point was a set of **axioms** that defined the environment of the project, **goals** that defined our hopes, and **design decisions** that helped us to reach those goals. Our implementation taught us that the goals are not easily reached. In particular, the issue of *simplicity* is quite complex; quests for simplicity in various areas often conflict with each other. The purpose of this paper is to explain the lessons we learned and motivate the steps we took while learning those lessons.

The axioms that constrained Charlotte's design were:

- **Charlotte will run on a multicomputer.** A *multicomputer* is a collection of conventional computers, each with its own memory, connected by a communications device. The tradeoffs between multicomputers and multiprocessors, which share memory, include scalability (multicomputers have a greater potential), grain of parallelism (multicomputers are suited only to large-grain parallelism), and expense (it seems less expensive to build a multicomputer).
- **Charlotte must support a wide variety of application programs.** Since the field of distributed computing is still young, we did not want to limit ourselves to client-server, pipeline, master-slave, or other communication paradigms and algorithm structures.

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

- **Policies and mechanisms will be clearly separated.** In order to experiment with policies, we did not want to embed them in the core of the operating system. Instead, we decided to place mechanisms in a *kernel* that is replicated on each machine. Policies are governed by *utility processes* whose location is generally irrelevant to the objects that they govern.

Our overall goals were simplicity and function:

- **Charlotte will provide adequate function.** The communication facilities of Charlotte should be appropriate to application programs covering various communication paradigms. This function should allow graceful degradation if some machines fail.
- **Charlotte will be simple.** Both the kernel and the utility processes should have this property. Simplicity has many dimensions. We intended Charlotte to be **minimal**, in the sense that it would not provide features that were not needed, and *efficient*, in the sense that the primitives could be executed quickly. We were also concerned that Charlotte be both **easily implemented** and **easily used.** As we will see, we were only partially successful in meeting these latter goals.

Our axioms and goals are not unique to Charlotte. The Accent system [Rashid81], Eden [Almes85], V kernel [Cheriton83], Medusa [Ousterhout80], StarOS [Jones79], Demos/MP [Powell83], Amoeba [Tanenbaum81], and a host of other operating systems have started with similar intentions. Other projects have tried to support distributed algorithms with languages instead of operating systems. Argus [Liskov83], NIL [Strom83], SR [Andrews82] and Ada [United States Department of Defense83] are examples of this approach.

Charlotte is unique in the design decisions we made. Our initial design decisions included the following:

- **Processes do not share memory.** This decision allows us to make inter-process communication completely location independent. It mirrors the fact that Charlotte runs on a multicomputer.
- **Communication is on reliable, symmetric, bi-directional paths named by capabilities.** Two-way paths are justified below. The use of capabilities (described more fully below) promotes an object-based model for clients and servers. Processes exercise control over who may send them messages. An action by one process cannot damage another, so long as the second takes basic precautions.

Capability-based naming also facilitates experimentation in migration for load balancing.

- **Absolute information is stored at each end of a communication path.** The information describes the machine, process, and path number of the other end of the path, and is stored in the kernel. Our choice of absolute information is not necessitated by the communication semantics we chose. The alternative (used in V kernel, for example), is to store hints at each end of the path, and to use broadcast as a fallback strategy when the hints fail.

The resulting operating system fulfills our goal of function and simplicity in some ways but not in others. The purpose of this paper is to see how the design decisions we made to satisfy our axioms and goals incurred implementation cost and required extra levels of software.

In section 2, we describe the IPC semantics of Charlotte. Although these semantics were intended to be simple, it turned out that supposedly orthogonal features interacted in complex ways. Section 3 shows how those semantics require a complex implementation. Application programmers found that while the IPC semantics make it possible to write highly concurrent programs, they also make it easy to commit subtle programming errors. Section 4 describes some of those errors and how they arise. To reduce the frequency of errors, we designed a programming language to regularize the use of Charlotte's primitives. Section 5 describes the language and how it makes programming easier and more secure. Section 6 describes the lessons that we gained from our experience.

2. Charlotte interprocess communication

The distinctive features of Charlotte IPC are duplex connections (*links*), dynamic link transfer, lack of buffering, nonblocking send and receive, synchronous wait, ability to cancel pending operations, and selectivity of receipt. We discuss each feature in turn. Detailed descriptions can be found elsewhere [Finkel83, Artsy86]. By way of summary, we start with a list of the Charlotte communication primitives.

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)" Post a send operation on a given link end, optionally enclosing another link end.

Receive (L : link; buffer : address; length : integer)

Post a receive operation on a given link end. L can be a specific link or an ''any link'' flag.

Cancel (L : link; d : direction)

Attempt to cancel a previously-started *Send* or *Receive* operation.

Wait (L : link; d : direction; var e : description)

Wait for an operation to complete. L can be a specific link or an ''any link'' flag. The direction can be *Sent*, *Received*, or either. The description returns the success or failure of the awaited operation, as well as its link, direction, number of bytes transferred, and the enclosed link (if any).

GetResult (L : link; d : direction; var e : description)

Ask for the information returned by *Wait*, but do not block if the operation has not completed. GetResult is a *polling* mechanism.

All calls return a status code. All but *Wait* are guaranteed to complete in a bounded amount of time.

The kernel matches *Send* and *Receive* operations. A match occurs if a *Send* and *Receive* request have been posted (and not cancelled) on opposite ends of the same link. Charlotte allows only one outstanding request in each direction on a given link. This restriction makes it impossible for an over-eager producer to overwhelm the kernel with requests. Completion must be reported by *Wait* before another similar request can be started. Buffers are managed by user processes in their own address spaces. Results are unpredictable if a process accesses a buffer between the starting of an operation and the notification of its completion.

Connections

Charlotte processes communicate by messages sent on *links*. A link is a software abstraction that represents a communications channel between two proceses. Each of the two processes has a capability to its end of the link. This capability confers the right to send and receive messages on that link. These rights cannot be duplicated, restricted or amplified. As we will see, they can be transferred, but there is at most

one capability in existence to any link end. Messages may be sent simultaneously in both directions on a single link.

We chose duplex links because our experience with Roscoe/Arachne [Solomon79], a predecessor to Charlotte, indicated several shortcomings of uni-directional links. First, client-server, master-slave, and remote-procedure-call situations all require information to flow in both directions. Even pipelines may require reverse flow for exception reporting. With uni-directional links, processes must manage link pairs, or reply links must be created, used once, and then discarded. Bi-directional links allow reverse traffic with no such penalty. Second, the kernel at a receiving end sometimes needs to know the location of the sending end, for example to warn the sending end that the receiving end has terminated or moved. In Arachne, Demos [Baskett77], and Demos/MP [Powell83], all of which use uni-directional links, the information stored at the receiving end of a link is not enough to find the sending ends. Bi-directional links offer the opportunity to maintain information at both ends to facilitate sending such information.

Link motion

A process can transfer possession of any link ends it holds by enclosing it in a message sent across a different link. The recipient of that message then gains possession of that link end. The sending process loses its ownership. While one end of a link is moving, the process at the other end may still post *Send* or *Receive* requests and can even move or destroy its end. Transfer of a link end is seen as an atomic action by processes. Link motion provides function (ability to change communication structures dynamically) while attempting to maintain simplicity (no effect on the stationary end). The goals of function and simplicity in the interface presented to processes was achieved at the expense of a rather complicated implementation, as discussed below.

Kernel buffering

An unlimited number of intermediate buffers would offer the highest degree of concurrency between senders and receivers. In practice, a system can only provide a finite number of buffers to store messages. A buffer pool requires deadlock prevention or detection techniques.

We felt that is is far easier to handle buffer allocation within the domain of each process instead of within th kernel. Therefore, the Charlotte kernel provides no buffering. This decision fits our simplicity

goals of minimality and ease of implementation. We will see in the next section that a small cache allows an efficient implementation. Since there are no kernel buffers, Charlotte does not need to place any limit on the size of messages.

Synchronization

Basic communication activities never block. The *Send* and *Receive* service calls initiate communication but do not wait for completion. In this way, a process may post *Send* or *Receive* requests on many links without waiting for any to finish. This design allows processes to perform useful work while communication is in progress. In particular, servers can respond to clients without fear that a slow client will block the server.

Posting a *Send* or *Receive* is synchronous (a process knows at what time the request was posted), but completion is inherently asynchronous (the data transfer may occur at any time in the future.) Charlotte provides three facilities for dealing with this asynchrony. First, versions of *Send* and *Receive* are available that block until they complete. Second, a process may explicitly wait for a *Send* or *Receive* to finish. Third, a process may poll the completion status of a *Send* or *Receive*. We also considered a fourth notification facility, software interrupts, but such a facility would have clashed with the blocking primitives we had, and it would have been difficult to use, because Charlotte provides no shared-memory synchronization primitives.

Request cancelling

A *Receive* or *Send* operation may be cancelled before it has completed. The *Cancel* request will fail if the operation has already been paired with a matching operation on the other end of the link.

Cancellation is useful in several situations. A server may grow impatient if its response to a client has not been accepted after a reasonable amount of time. A sender may discover a more up-to-date version of data it is trying to *Send*. A receiver may decide it is willing to accept a message that requires a buffer larger than the one provided by its current *Receive*. A server that keeps *Receives* posted as a matter of course may decide it no longer wants messages on some particular link. These last two scenarios arise in the run-time support routines for the LYNX language, described in section 5.

Message filtering

The *Receive* and *Wait* requests can specify a single link end or all link ends owned by the caller. We considered allowing more general sets of link-ends but rejected this feature for reasons of simplicity. *Wait* can also specify whether a Send or Receive event (or either) is to be awaited.

3. Implementation

Charlotte is implemented on the Crystal multicomputer [DeWitt84], a collection of Digital VAX–11/750s connected by a high-speed token ring. Charlotte resides above a communication package called the *nugget* [Cook83], which provides a reliable, packetized, inter-machine transmission service. Charlotte's kernel implements the abstractions of processes and links. In order to provide the facilities described earlier, kernels use a lower-level communication protocol. Significant events for this protocol include messages received from remote kernels and requests from local processes.

The protocol can be described in terms of scenarios of increasing complexity. In general, the kernel attempts to match *Send* and *Receive* requests on opposite ends of a link. When it succeeds in doing so it transfers the contents of the message and moves the enclosed link-end, if any.

The simplest case arises when a *Send* is posted with no enclosure, and the matching *Receive* is already pending. In this case, the sending kernel transmits one packet to the receiving kernel and the latter responds with an acknowledgment. If the matching *Receive* is posted after the packet arrives, the message may still be held in a cache on the receiving kernel, so the acknowledgement can still be sent. If the message is no longer in the cache, the receiving kernel asks the sending kernel to retransmit it.

Extra complexity is introduced by messages too large to fit in a single packet. (The nugget imposes a limit of about 2KB.) In this case the receiving kernel acknowledges the first packet, the sending kernel sends the rest, and the receiving kernel returns a single acknowledgement for all.

Still further complexity is introduced by cancelling *Send* and *Receive* requests. The request might be in any of several states, such as pending, matched, in transit, aborted, or completed. An already-matched transaction might even be cancelled by both ends at once.

Link movement and destruction introduce added complexity, especially when both occur at the same time. Link destruction is straightforward if the link is idle. If the remote end has a request pending, the situation becomes slightly more complex. If that request is a *Send* with an enclosed link-end, or if the remote end is itself in motion, the situation becomes even more complex. Such cases will also occur when a process terminates unexpectedly. We discuss these complex scenarios and present the protocol elsewhere [Artsy84].

As we mentioned earlier, Charlotte attempts to keep both ends of a link consistent. Link movement therefore requires that a third party (the kernel of the other end of the link that is moving) be informed. That third party may have a pending *Send* or *Receive* of its own. The protocol must also deal with both ends moving simultaneously.

Our initial simple set of operations turned out to have complex interrelations. An action taken at one end of a link can happen when the other end is in any state. Since *Send* and *Receive* do not block the caller, the local end of the link may also be in many states. A significant amount of co-ordination is needed to make sure each end of the link holds consistent information. The fact that either end can destroy a link and that communication travels in both directions means that propagation of information can interfere in arbitrary ways with communication in progress.

We designed the implementation so that all cases, both simple and complex, could be handled in a regular manner. The protocol is managed by a hand-built, table-driven finite state automaton. The states of the automaton reflect the status of a link. The inputs to the automaton are requests from processes and packets from remote kernels. We enumerated cases and manually simulated them in an attempt to verify the correct behavior of the protocol. The kernel uses twenty message types, six of which represent process requests, and the rest of which represent kernel-kernel messages. As is true in other systems [Bochmann77] the number of states is large. To keep the complexity manageable we built four independent automata for different functions: *Send*, *Receive*, *Destroy*, and *Move*. These automata interact in only a few cases. Destroying a link, for example, affects its *Send* state. Cancellation is implemented in the *Send* and *Receive* automata.

We also reduced the number of states with a method described by Danthine [Danthine76] and others [Bochmann78] in which some information is encoded in global variables. The variables are only consulted

for particularly complex cases.

Our automata have approximately 250 non-error entries, for each of which an action is prescribed. Often, same action applies to several different entries; the total number of actions is about 100. The simplest actions consist of a single operation, such as sending a completion acknowledgment. The most complex action checks five variables and selects one of several operations.

The kernel itself is implemented as a collection of non-preemptable Modula processes [Finkel83b], which we call *tasks* to distinguish them from user and utility processes. These tasks communicate via queues of work requests. The **Automaton** task implements all four automata. All process requests are first verified by the **Envelope** task. Communication requests are then forwarded to the automaton's work queue. Two tasks deal with information flow to and from the nugget. Other tasks are responsible for maintaining the clock, collecting statistics, and checking to make sure that other nodes are alive. Processes run only when kernel tasks have nothing to do.

This division of labor simplifies the implementation. We have found the kernel relatively easy to maintain. Most errors can be isolated to a specific action. Modifications are usually isolated to just a few actions. In particular, we implemented process migration as an incremental modification of Charlotte without needing to modify the automata in any substatial way [Artsy86b].

On the other hand, bugs continue to be discovered as less-frequently used states are entered. These bugs are usually not protocol errors, but rather omission of necessary record-keeping details. The complete kernel-kernel protocol is almost beyond comprehension by a single person. In this sense, Charlotte has failed to achieve simplicity.

We have found that our originally simple semantics have required occasional revision and elucidation as we encountered new situations. The resulting semantics are quite complex. Occasionally, we decided to disallow certain combinations instead of defining the semantics. For example, we prohibit a *Receive*(Any) in the presence of any other *Receive*.

A major lesson that we draw from our implementation is that it is difficult to juggle simplicity and function, especially when the starting point is a set of seemingly independent axioms defining the process-kernel communications interface. One danger is that implementing this interface will introduce complexi-

ties, some of which must be reflected back to the interface design. Another danger is that the interface will not fit the needs of actual applications. Other starting points have different dangers. For example, starting with a language design (such as NIL or Argus) is more likely to serve applications well, but is riskier with respect to efficient implementation.

Further lessons were learned by building the utility processes that control policy on Charlotte. The next section describes what we found.

4. Programming in Charlotte

We all learn when writing programs for the first time that it is almost impossible to avoid bugs. The problem appears to be much worse in a distributed environment. In addition to logical errors within a process, there are also errors in the way processes deal with each other. In particular, an *ordering error* is an error caused by the relative order in which operations occur [Gordon85]. In Charlotte, the principal ordering error that processes are likely to commit is using a receive buffer or modifying a send buffer before the associated *Receive* or *Send* operation has finished.

We found that writing utility processes is not easy. Timing errors are only part of the problem. Sequential languages with service calls for communication services are not particularly suited to writing these programs. Not only is type checking a problem across modules (like other operating systems, Charlotte does not try to prevent or detect type mismatches), but flow of control within a single process can be a headache. A server that manages several clients has to maintain several conversations, each with its own state. One way to deal with this complexity is to create a separate worker process for each active client. Each worker is restricted to one conversation and may use blocking communication. This approach has been described by Liskov, Herlihy, and Gilbert [Liskov83b,Liskov84]. Charlotte (like other similar operating systems) does not promote this style for several reasons. First, process initiation is expensive, mostly because policy is involved (on which machine should the process be started, and where in memory), so a policy process must be consulted. Second, the workers cannot share space, since Charlotte does not provide shared memory. (It would have been helpful if Charlotte provided light-weight processes that share memory, in addition to its heavy-weight ones, which don't. Argus [Liskov83] and Amoeba [Tanenbaum81] provide such a facility.)

Instead of using multiple workers, Charlotte utilities are written in a conventional sequential language (a Modula subset) using non-blocking communication in a single server. This approach leads to several serious problems.

- Servers devote a considerable amount of effort to packing and unpacking message buffers. The standard technique uses type casts to overlay a record structure on an array of bytes. Program variables are assigned to or copied from appropriate fields of the record. The code is awkward at best and depends for correctness on programming conventions that are not enforced by the compiler. Errors due to incorrect interpretation of messages have been relatively few, but they are very hard to find.
- Every kernel call returns a status variable whose value indicates whether the requested operation succeeded or failed. Different sorts of failures result in different values. A well-written program must inspect every status variable and be prepared to deal appropriately with every possible value. It is not unusual for 25 or 30% of a carefully-written server to be devoted to error checking and handling. Even an ordinary client must explicitly handle error returns, if only to terminate when a problem occurs.
- Conversations between servers and clients often require a long series of messages. A typical conversation with a file server, for example, begins with a request to open a file, continues with an arbitrary sequence of read, write, and seek requests, and ends with a request to close the file. The flow of control for a single conversation could be described by simple, straight-line code except for the fact that the server cannot afford to wait in the middle of that code for a message to be delivered. We haven't discovered a programming style that makes the explicit interleaving of separate conversations easy to read and understand.

This last problem is probably the most serious, at least for writing utility processes. In order to maximize concurrency and protect themselves from misbehaving clients, Charlotte servers break the code that manages a conversation into many small pieces, separated by requests for communication. They invoke the pieces individually so that conversations can be interleaved. Every Charlotte server shares the following overall structure:

```
begin
  initialize
  loop
    wait for a communication request to complete
    find the conversation waiting for this communication
    case request.type of
      A :
        restore state of conversation
        compute
        start new request
        save state
      B :
        ...
    end case
  end loop
end.
```

The flow of control for a typical conversation is hidden by the global loop. This style of programming is hard to write and harder to read.

The lesson that we learn from this discussion is that providing adequate function does not automatically make facilities easy to use. It is natural for operating systems to provide communication facilities through service calls, but it is not necessarily natural for programs to operate at that level. The hardest problems seem to arise in servers. Clients are more straightforward to write, since the server-specific protocol can be packaged into a library routine that makes communication look like procedure calls (at the expense of blocking during all calls to servers).

There are two ways out of the difficulty. One is to provide a higher level of service in the kernel, such as light-weight processes. We suspect this alternative will be complex and difficult to implement. The other is to layer a higher-level interface upon the communication primitives. The next section describes this direction.

5. The LYNX programming language

In order to overcome the difficulties described above, we have developed the LYNX language. It is described in full detail elsewhere [Scott84, Scott85]. LYNX differs from most other distributed languages we have surveyed [Scott84b] in three major areas:

- **Processes and Modules.** Processes and modules in LYNX reflect the structure of a multicomputer. Modules may nest, but only within a machine; no module can cross the boundaries between

machines. Each outermost module is inhabited by a single process. Processes share no memory. They are managed by the operating system kernel and execute in parallel. Multiple threads of control within a process are managed by the language run-time system, but there is no pretense of parallelism among them.

- **Communication Paths and Naming.** LYNX provides Charlotte links as first-class language objects. The programmer has complete run-time control over the binding of links to processes and binding of names to links. The resulting flexibility allows links to be used for reconfigurable, type-checked connections between very loosely-coupled processes — processes written and loaded at widely disparate times.
- **Syntax for Message Receipt.** Messages in LYNX may be received explicitly by any thread of control. They may also be received implicitly, creating new threads that execute entry procedures. Processes can decide at run time which approach(es) to use when, and on which links.

Each LYNX process begins with a single thread of control. It can create new threads locally or can arrange for them to be created in response to messages from other processes. Separate threads do *not* execute in parallel; a given process continues to execute a given thread until it blocks. It then takes up some other thread where it last left off. If all threads are blocked for communication, then the process waits for a message to be sent or received.

In a server, each thread of control is used to manage a single **conversation** with a client. Conversations may be subdivided by creating new threads at inner levels of lexical nesting. The activation records accessible at any given time will form a tree, with a separate thread corresponding to each leaf.

A link variable in LYNX accesses one end of a link, much like a pointer accesses an object in Pascal [Jensen74]. The distinguished value "nolink" is the only link constant. Built-in functions allow new links to be created and old ones to be destroyed. (Neither end of a destroyed link is usable.)

Objects of any data type can be sent in messages. If a message includes link variables or structures containing link variables, then the link ends referenced by those variables are moved to the receiving process. Link variables in the sender that refer to those ends become dangling references; a runtime error results from any attempt to use them.

Message transmission looks like a remote procedure call; the sending thread of control dispatches a **request** message and waits for a **reply** from the receiver. An active thread can serve a request by executing an Ada-like [United States Department of Defense83] **accept** statement. A process can also arrange to receive requests implicitly by **binding** a link to an entry procedure.

Bind and **unbind** are executable commands. A link may be bound to more than one entry. The bindings need not be created at the same time. A bound link can even be used in subsequent accept statements. These provisions make it possible semantically for separate threads to carry on independent conversations on the same link at approximately the same time.

When all threads in a process are blocked, the run-time support routines attempt to receive a request on any of the links for which there are bindings or outstanding *accepts*. The **operation name** contained in the message is matched against those of the *accepts* and the bound entries to decide whether to resume an existing thread or create a new one. Bindings or *accepts* that cause ambiguity are treated as run-time errors.

LYNX enforces structural type equivalence on inter-process calls. A novel mechanism for self-descriptive messages [Scott84c] allows the checking to be performed efficiently on a message-by-message basis. An exception handling mechanism 1) permits recovery from errors that arise in the course of message passing, and 2) allows one thread to interrupt another.

Our initial experience with LYNX has shown several results, some of which were unexpected:

- It is far easier to write servers in LYNX than in a sequential language with calls to Charlotte primitives. The LYNX code is also easier to read and understand. LYNX implementations of servers use less than half as many lines of source code.
- Despite the fact that much of the design of LYNX was motivated by the primitives of Charlotte, the actual process of implementation proved to be quite difficult. A paper implementation of LYNX built on SODA, a "Simplified Operating system for Distributed Applications," was in some ways considerably simpler.

Problems with the implementation for Charlotte were two-fold. First, LYNX requires greater selectivity than Charlotte provides for choosing an incoming message. Second, LYNX permits an arbitrary

number of links to be enclosed in a message, while Charlotte supports only one. The simplicity that the Charlotte design attempted to capture turned out to provide not quite the right function.

On the other hand, LYNX has been a success. We have used it to reimplement most of the utility processes as well as a host of distributed applications, including numerical applications (the Simplex method), AI techniques (ray tracing, Prolog), data structures (nearest neighbor search in k-d trees, B+ trees), and graph algorithms (spanning tree, travelling salesman) [Finkel86, Finkel86b]. The Charlotte link concept, as represented in LYNX, turns out to be a valuable way to represent resources and algorithm structure. This vindicates our original choice of bi-directional links.

6. Conclusions

We have learned the following important lessons from our experiences.

- **Simple primitives interact in complex ways.** At first glance, separating the posting of a *Send* from the notification of its completion might be expected to simplify the kernel by lowering the level of its primitive operations. This separation, however, introduces complexities when taken together with the ability to destroy a link. Similarly, the ability to pass a link in a message seems to be a simple idea, but it becomes complex if both ends are in motion at the same time.
- **It is not easy to make use of asynchronous primitives.** The combination of nonblocking *Send /Receive* and the lack of buffering in the kernel makes it easy to overwrite a buffer.
- **Appropriate higher-level tools mitigate the programming problems.** We have mentioned LYNX as one such tool. In addition, library packages that understand how to talk to servers make writing clients much easier. We have also implemented a *connector* process that supervises the initial interconnection of cooperating processes. The Matchmaker service in Accent [Rashid81] serves a similar function.
- **Absolute distributed information is hard to maintain.** Absolute, up-to-date, consistent, distributed information can be more trouble than it is worth. It may be considerably easier to rely on a system of hints, so long as they usually work, and so long as we can get correct information (perhaps at additional cost) when they fail.

- **Screening belongs in the application layer.** Every reliable protocol needs top-level acknowledgments [Saltzer84]. 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. This trick 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. For implementing LYNX, the descriptive mechanisms of Charlotte were not rich enough.
- **Middle-level primitives are usually at the wrong level.** From the point of view of the language implementer, the application developer, and even the operating-system designer, the "ideal operating system" probably lies at one of two extremes: it either supports the language or application directly, or it provides only a minimal sufficient set of primitives. Providing direct support to one particular application (like a programming language) is likely to make the kernel less appropriate for other applications.

Our initial decision to provide links was motivated by our perception that they would regularize communication, thus making programs easy to write. We did not foresee the complexity of implementing links. We managed to create a fairly efficient implementation, but the underlying protocol has an enormous number of states. The process-migration protocol [Artsy86b], which supplements the IPC protocol, did not add significant complexity. Links *do* regularize communication, but the IPC semantics we chose do not lend themselves to programming ease. The problem was especially severe when we wrote server processes. Although we had enough power to write efficient servers, it was hard to write the programs correctly. Writing programs whose IPC is restricted to operating-system primitives is like writing programs whose control structures are restricted to the *goto* statement. We devised LYNX to provide a more comfortable medium. We found that Charlotte's IPC, although sufficiently powerful, was not sufficiently low-level to implement LYNX easily. On the other hand, LYNX vindicates our original choice of bi-directional links. They provide an elegant way to describe resources.

On the whole, Charlotte has lived within its constraints and achieved its goals. Charlotte provides a functioning framework for writing distributed applications, and many projects are underway in designing such applications. LYNX has been valuable beyond the multicomputer framework in which it was

developed; it has been successfully ported to the Butterfly machine [Scott86]. Given LYNX, library routines, and our connector facility, it is fairly easy to write correct and intelligible applications.

7. References

Almes85.
Almes, G. T., A. P. Black, E. D. Lazowska, and J. D. Noe, "The Eden System: A Technical Review," *IEEE Transactions of Software Engineering* **SE-11**(1) pp. 43-59 (January 1985).

Andrews82.
Andrews, G. R., "The Distributed programming language SR — mechanisms, design and implementation," *Software — Practice and Experience* **12** pp. 719-753 (1982).

Artsy84.
Artsy, Y., 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).

Artsy86.
Artsy, Y., H-Y Chang, and R. Finkel, "Interprocess communication in Charlotte," *IEEE Software*, (Accepted subject to revision) (July 1986).

Artsy86b.
Artsy, Y., H-Y Chang, and R. Finkel, "Processes migrate in Charlotte," Computer Sciences Technical Report (in preparation), University of Wisconsin–Madison (August 1986).

Baskett77.
Baskett, F., J. H. Howard, and J. T. Montague, "Task communication in Demos," *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, pp. 23-31 (November 1977).

Bochmann77.
Bochmann, G. V. and J. Gescei, G. V. Bochmann, and C. A. Sunshine, "Formal methods in communication protocol design," *IEEE Transactions on Communication* **Com-28**(4) pp. 624-631 IFIP, North-Holland, (April 1980).

Bochmann78.
Bochmann, G. V., "Finite State Description of Communication Protocol," *Computer Networks* **2** pp. 361-372 (1978).

Cheriton83.
Cheriton, D. R. and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pp. 128-139 (In *ACM Operating Systems Review* 17:5) (10-13 October 1983).

Cook83.
Cook, R., R. Finkel, D. DeWitt, L. Landweber, and T. Virgilio, "The Crystal nugget: Part I of the first report on the Crystal project," Technical Report 499, Computer Sciences Department, University of Wisconsin (April 1983).

Danthine76.
Danthine, A. and J. Bremer, "An Axiomatic Description of the Transport Protocol of Cyclades," *Professional Conference on Computer Networks and Teleprocessing*, (March 1976).

DeWitt84.
DeWitt, D., R. Finkel, and M. Solomon, "The Crystal multicomputer: Design and implementation experience," Technical Report 553 (To appear, IEEE Transactions on Software Engineering) , University of Wisconsin–Madison Computer Sciences (September 1984).

Finkel83.
Finkel, R., M. Solomon, D. DeWitt, and L. Landweber, "The Charlotte Distributed Operating System: Part IV of the first report on the crystal project," Technical Report 502, University of Wisconsin–Madison Computer Sciences (October 1983).

Finkel83b.
Finkel, R., R. Cook, D. DeWitt, N. Hall, and L. Landweber, "Wisconsin Modula: Part III of the first report on the crystal project," Computer Sciences Technical Report #501, University of Wisconsin–Madison (April 1983).

Finkel86.
Finkel, R. A., A. P. Anantharaman, S. Dasgupta, T. S. Goradia, P. Kaikini, C-P Ng, M. Subbarao, G. A. Venkatesh, S. Verma, and K. A. Vora, "Experience with Crystal, Charlotte, and Lynx," Computer Sciences Technical Report #630, University of Wisconsin–Madison (February 1986).

Finkel86b.
Finkel, R. A., B. Barzideh, C. W. Bhide, M-O Lam, D. Nelson, R. Polisetty, S. Rajaraman, I. Steinberg1, and G. A. Venakatesh, "Experience with Crystal, Charlotte, and Lynx: Second Report," Computer Sciences Technical Report #649, University of Wisconsin–Madison (July 1986).

Gordon85.
Gordon, A. J., *Ordering errors in distributed programs* (Ph.D. thesis) (May 1985).

Jensen74.
Jensen, K. and N. Wirth, "Pascal: User Manual and Report," *Lecture Notes in Computer Science*, (18) Springer-Verlag, (1974).

Jones79.
Jones, A. K., R. J. Jr. Chansler, I. Durham, K. Schwans, and S. R. Vegdahl, "StarOS, a multiprocessor operating system for the support of task forces," *Proc. 7th Symposium on Operating Systems Principles*, pp. 117-127 (December 1979).

Kepecs84.
Kepecs, J. H. and M. H. Solomon, "SODA: A simplified operating system for distributed applications," *Third Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, (Aug 27-29, 1984).

Liskov83.
Liskov, B. and R. Scheifler, "Guardians and actions: Linguistic support for robust, distributed programs," *ACM TOPLAS* **5**(3) pp. 381-404 (July 1983).

Liskov83b.
Liskov, B. and M. Herlihy, "Issues in process and communication structure for distributed programs," *Proceedings of the Third IEEE Symposium on Reliability in Distributed Software and Database Systems*, pp. 123-132 (October 1983).

Liskov84.
Liskov, B., 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).

Ousterhout80.
Ousterhout, J. K., D. A. Scelza, and S. S. Pradeep, "Medusa: An experiment in distributed operating system structure," *CACM* **23**(2) pp. 92-105 (February 1980).

Powell83.
Powell, M. L. and B. P. Miller, "Process migration in DEMOS/MP," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, pp. 110-118 (In *ACM Operating Systems Review* 17:5) (10-13 October 1983).

Rashid81.
Rashid, R. F. and G. G. Robertson, "Accent: A communication oriented network operating system kernel," *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pp. 64-75 (14-16 December 1981).

Saltzer84.
Saltzer, J. H., D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM TOCS* 2(4) pp. 277-288 (November 1984).

Scott84.
Scott, M. L. and R. A. Finkel, "LYNX: A dynamic distributed programming language," *1984 International Conference on Parallel Processing*, (August, 1984).

Scott84b.
Scott, M. L., "A framework for the evaluation of high-level languages for distributed computing," Computer Sciences Technical Report #563, University of Wisconsin–Madison (October 1984).

Scott84c.
Scott, M. L. and R. A. Finkel, "A simple mechanism for type security across compilation units," Computer Sciences Technical Report #541, University of Wisconsin–Madison (May 1984).

Scott85.
Scott, M. L., "Design and implementation of a distributed systems language," Ph. D. Thesis, Technical Report #596, University of Wisconsin–Madison (May 1985).

Scott86.
Scott, M. L., "Lynx reference manual," BPR 7, Computer Science Department, University of Rochester (March 1986).

Solomon79.
Solomon, M. H. and R. A. Finkel, "The Roscoe distributed operating system," *Proc. 7th Symposium on Operating Systems Principles*, pp. 108-114 (December 1979).

Strom83.
Strom, R. E. and S. Yemini, "NIL: An integrated language and system for distributed programming," *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, pp. 73-82 (In *ACM SIGPLAN Notices* 18:6 (June 1983)) (27-29 June 1983).

Tanenbaum81.
Tanenbaum, A. S. and S. J. Mullender, "An overview of the amoeba distributed operating system," *ACM Operating Systems Review* **15**(3) pp. 51-64 (July 1981).

United States Department of Defense83.
United States Department of Defense,, "Reference Manual for the Ada Programming Language," ANSI/MIL-STD-1815A-1983 (February 1983).