# Experience with Charlotte: Simplicity and Function in a Distributed Operating System

RAPHAEL A. FINKEL, MICHAEL L. SCOTT, MEMBER, IEEE, YESHAYAHU ARTSY, AND HUNG-YANG CHANG

*Abstract*—This paper presents a retrospective view of the Charlotte distributed operating system, a testbed for developing techniques and tools to solve computation-intensive problems with large-grain parallelism. The final version of Charlotte runs on the Crystal multicomputer, a collection of VAX-11/750 computers connected by a local-area network. The kernel/process interface is unique in its support for symmetric, bidirectional communication paths (called links), and synchronous nonblocking communication.

Our experience indicates that the goals of simplicity and function are not easily achieved. Simplicity in particular has dimensions that conflict with one another. Although our design decisions produced a high-quality environment for research in distributed applications, they also led to unexpected implementation costs and required high-level language support.

We learned several lessons from implementing Charlotte. Links have proven to be a useful abstraction, but our primitives do not seem to be at quite the right level of abstraction. Our implementation employed finite-state machines and a multitask kernel, both of which worked well. It also maintains absolute distributed information, which is more expensive than using hints. The development of high-level tools, particularly the Lynx distributed programming language, has simplified the use of kernel primitives and helps to manage concurrency at the process level.

*Index Terms*—Charlotte, Crystal, distributed computing, kernel interface design, links, Lynx, message passing.

## I. Introduction

CHARLOTTE is a distributed operating system in production use at the Department of Computer Sciences of the University of Wisconsin—Madison [4], [5]. Charlotte is intended as a testbed for developing techniques and tools to exploit large-grain parallelism in computation-intensive problems. Charlotte was constructed over the course of approximately five years, going through several distinct versions as the underlying hardware and our ideas for implementation changed. The final version runs on the Crystal multicomputer [17], a collection of 20 VAX-11/750 computers connected by an 80 Mbit/second Proteon token ring. This paper presents a retrospective view of the Charlotte project.

Although it seems clearer in hindsight than it was in the early stages, we now regard our work as the result of 1) the *axioms* that defined the available design space, 2) the *goals* that provided direction, and 3) the *design decisions* that established the final structure. Our experience indicates that the goals of simplicity and function are difficult to attain simultaneously. Simplicity is particularly troublesome: quests for simplicity in different areas of a project may conflict with one another. The purpose of this paper is to explain the lessons we learned from Charlotte and to motivate the steps we took while learning those lessons.

The axioms for Charlotte were as follows:

- *The underlying hardware will be a multicomputer.* A multicomputer is a collection of conventional computers (called nodes), each with its own memory, connected by a communicative device. The tradeoffs between multicomputers and multiprocessors, in which memory is shared, include scalability (multicomputers have greater potential), grain of parallelism (multicomputers are suited only to a large-grain parallelism), and expense (multicomputers appear to be more economical). At the time our project began, the department was heavily committed, both in research orientation and hardware resources, to parallel computing without shared memory.
- *The project will support a wide variety of message-based applications.* The design of distributed algorithms has been an important area of research in our department for many years. Our prospective user community would not have been willing to adopt a single programming language or a single paradigm for process interaction (client-server, master–slave, or pipeline, for example). Ideally, we would have liked to support a variety of shared-memory paradigms as well, but our commitment to a multicomputer environment made such support impractical.
- *Policies and mechanisms will be separated clearly.* In addition to supporting distributed applications, it was imperative that our work permit experimentation with distributed systems software. The Charlotte kernel needed to provide sufficient mechanisms to make good use of the machine, but could not afford to embed policies in the core of the operating system. A fundamental assumption, then, was that the majority of operating system services would be provided by user-level server processes, outside the (replicated) kernel.

Manuscript received February 27, 1987; revised October 30, 1987.

R. A. Finkel is with the Department of Computer Science, University of Kentucky, Lexington, KY 40506.

M. L. Scott is with the Department of Computer Science, University of Rochester, Rochester, NY 14627.

Y. Artsy is with Digital Equipment Corporation, 550 King Street, Littleton, MA 01460.

H.-Y. Chang is with the IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598.

IEEE Log Number 8927379.

We have labeled the preceding items "axioms" because, in retrospect, it is clear that they were never questioned in the course of the Charlotte project. To a large extent, they were imposed by outside factors. They were also objective enough that little interpretation was required. By contrast, our goals were much more vaguely stated:

- *Charlotte will provide adequate function.* The facilities available to application programs must be expressive enough to support the needs of our user community (axiom 2). In its attempt to provide a pleasant virtual machine, the kernel must not hide too much of the power of the underlying hardware. Graceful degradation of service must occur when individual nodes of the multicomputer fail.
- *Charlotte will be simple.* Simplicity is largely aesthetic, although it has a number of more concrete 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 it did provide would require little code and could be executed quickly. We were also concerned that Charlotte be both *easily implemented* and *easily used.* We hoped to describe its primitives with short, *concise semantics*. As discussed below, we were successful in only some of these dimensions.

Our axioms and goals are not unique to Charlotte. Accent [30], Amoeba [28], Demos/MP [27], Eden [2], V [13], 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. Ada [40], Argus [26], NIL [39], and SR [3] are examples of this approach. It is the combination of design decisions we made in building Charlotte that makes our work unique. Our earliest, most influential decisions are summarized below. These set the stage for the many smaller decisions described in Section II.

- *Processes do not share memory*, even within a single node of the machine. This decision allows us to make interprocess communication (IPC) completely location independent. It mirrors the fact that Charlotte runs on a multicomputer.
- *Communication is on reliable, symmetric, bidirectional links named by capabilities.* Two-way links are justified below. The use of capabilities (described more fully below) provides a useful abstraction for distributed resources. 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 with migration for load balancing.
- *Communication is nonblocking, but synchronous.* A server process must often have conversations in progress with a large number of clients at once. It is imperative that sending and receiving be nonblocking operations. Unfortunately, it is also imperative that processes know when communication fails. The kernel cannot always be trusted to deliver a message successfully, and a process that continues execution after making a request may be in an arbitrary state when problems arise. Charlotte addresses these concerns by allowing a process to discover the status of its messages explicitly, at a time of its own choosing.

A final version of Charlotte fulfils our goals of function and simplicity in some ways but not in others. We are hopeful that our experience will prove useful to future designers of similar systems, who may wish to emulate our successes and avoid our mistakes. We describe the Charlotte IPC semantics in Section II. Although these semantics were intended to be simple, supposedly orthogonal features were found to interact in unexpected ways. The kernel/process interface comes close to our goal of *minimality*, but the implementation is large and complex. Careful efforts to keep the kernel modular and structured made it relatively *easy to build* and maintain, despite its size, but the goal of *efficiency* suffered badly in the process. We describe the implementation in Section III.

Above the level of the kernel/process interface, experience with the first generation of server processes convinced us that high-level language support would be required to make Charlotte *easy to use*. While the IPC semantics made it possible to write highly concurrent programs, they also required a distasteful amount of user code and make it easy to commit subtle programming errors. Section IV describes some classes or errors and explains how they arise. Section V describes the language we developed to regularize the use of Charlotte primitives, handle exceptional conditions, and manage concurrent conversations. Section VI describes the lessons we have learned from our experience.

## II. Charlotte Interprocess Communication

For reference purposes, we begin this section with a summary of the most important Charlotte communication primitives. The list may be skimmed on first reading and then consulted when appropriate later. Complete descriptions of the Charlotte kernel/process interface can be found in other papers [4], [5], [22].

*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 request on a given link end, optionally enclosing another link end.

*Receive* (L : link; buffer : address; length : integer)
Post a receive request 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-posted *Send* or *Receive* request. The attempt will fail if the request has already completed, even if it has not yet been awaited.

*Wait* (L : link; d : direction; var e : description)
Wait for a request 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 request, 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 request has not completed. GetResult is a polling mechanism.

The kernel matches *Send* and *Receive* requests. A match occurs if a *Send* and a *Receive* have been posted (and not canceled) 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 through *Wait* or *GetResult* before another similar request can be posted. Buffers are managed by user processes in their own address spaces. Results are unpredictable if a process accesses a buffer between posting a request and receiving notification of its completion.

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

### A. Connections

Charlotte processes communicate with messages sent on *links*. A link is a software abstraction that represents a communication channel between a pair of processes. Messages may be sent in either direction on a link. They can even be sent simultaneously in both directions. Each process has a capability to its end of the link. This capability confers the right to send and receive messages on the link. These rights cannot be duplicated, restricted or amplified. They can be transferred to another process (see below), but the kernel guarantees that only one capability for each link end exists at a given time.

Among other things, the status codes returned form kernel calls allow a process to determine when one of its links has been destroyed at the other end. The kernel destroys all the links connected to a process automatically when the process terminates. It also destroys all the links connected to processes on a particular node when it detects that the node has crashed.

The decision to use duplex links was something of an experiment. Experience with Arachne (Roscoe) [38], a predecessor to Charlotte, indicated several shortcomings of unidirectional links, in which messages can be sent in one direction only. 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 unidirectional links, processes must manage link pairs or must create reply links to be used once and then discarded. Bidirectional 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(s), to warn them, for example, that the receiving end has moved or been destroyed. In Arachne, Demos [9], and Demos/MP [27], all of which use unidirectional links, the information stored at the receiving end of a link is not enough to find the sending ends. Bidirectional links offer the opportunity to maintain information at both ends about the location of the other.

A Charlotte process can transfer possession of one of its link ends to another process by enclosing the end in a message on another link. The receiver of the message gains possession of the moving end. The sending process loses its possession. 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 an atomic operation from the user's point of view. This atomicity, particularly in the presence of canceled requests and simultaneously moving ends, was achieved at the expense of a rather complicated implementation, as discussed in Section III.

### B. Buffering and Synchronization

An unlimited number of kernel buffers would offer the highest degree of concurrency between senders and receivers. In practice, a message-passing system can only provide a finite amount of storage. Management of a pool of buffers requires flow control and deadlock prevention or recovery. Rather than accept the resulting complexity, we decided in Charlotte to manage buffers in user space. This decision is in keeping with the goals of *minimality* and *ease of implementation*. As described in the following section, a cache of (semantically invisible) buffers in the kernel permits an *efficient* implementation as well. Moreover, the use of user-provided buffers allows Charlotte to handle messages of arbitrary size.

### C. Synchronization

*Wait* is the only communication primitive that blocks. *Send* and *Receive* initiate communication but do not wait for completion. A process may therefore post *Send* or *Receive* requests on many links before waiting for any to finish. It may also perform useful work while communication is in progress. Servers in particular need not fear that one slow client will compromise the service provided to others.

Posting a *Send* or *Receive* is synchronous (a process knows the time at which the request was posted), but completion is inherently asychronous (the data transfer may occur at any time in the future). Charlotte allows a user process to poll for completion status, with *GetResult*, or to block, with *Wait*, until that status is available. For processes that want to wait immediately after posting a request, there are also combined *Send/Wait* and *Receive/Wait* kernel calls (not listed in Section II) that avoid the extra context switches. We consider a mechanism for software completion interrupts, but had no applications in mind for which the additional functionality would have justified the complexity of semantics and implementation. By contrast, the ability to cancel an outstanding *Send* or *Receive* request appeared to be useful in several common situations.

A server may wish to cancel a *Send* if its message to a client has not been accepted after a reasonable amount of time. Likewise, a process may wish to cancel a *Send* when it discovers a more up-to-date version of the data it is trying to transmit. 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 V. Naturally, a *Cancel* request will fail if its *Send* or *Receive* has already been paired with a matching request at the other end of the link.

### D. Message Screening

The *Receive* and *Wait* requests can specify a particular link end, or can indicate than any end will do. *Wait* can specify whether a *Send* request, a *Receive* request, or *Either* is awaited. We considered allowing more general sets of link ends, but as with software interrupts had no applications in mind for which the added complexity was essential. The ability to specify an arbitrary set of ends would have had a negative impact on all five of our measures of simplicity.

Since only a single *Send* and a single *Receive* can be outstanding on a given link end, the combination of link number and direction suffices to specify a request to be canceled, polled, or awaited. We considered a scheme in which *Send* and *Receive* would each return a request identifier to be used when referring to the pending request. Such a scheme would have made it easier to support multiple requests (for double buffering, for example). It would also have simplified the specification of sets of requests for *Wait* had we permitted them. It was our original impression that the provision of request identifiers would have increased both the complexity of the kernel/process interface and the size and overhead of the kernel. In hindsight, it appears that this impression was mistaken; request identifiers would probably have made life easier for both the user and the kernel.

## III. Implementation

On the Crystal multicomputer [17], Charlotte resides above a communication package called the *nugget* [15], which provides a reliable, packetized, intermachine transmission service. Charlotte's kernel implements the abstractions of processes and links. In order to provide the facilities described in the previous section, copies of the kernel on separate nodes communicate with a lower-level protocol. Significant events for this protocol include messages received from remote kernels and requests from local processes.

### A. Protocol

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 in the receiving kernel, so the acknowledgment can still be sent. If the message is no longer in the cache, the receiving kernel asks the sending kernel to retransmit it.

Very large messages may require multiple packets, since the nugget imposes a maximum size of approximately 2K bytes. The receiving kernel acknowledges the first packet when the receiving process is ready. The sending kernel then transmits the remaining packets and the receiving kernel returns a single acknowledgment for all.

An attempt to cancel a *Send* or *Receive* request may find that request in any of several states, such as pending, matched, in transit, aborted, or completed. Likewise, an attempt to move or destroy a link may find the other end in any of a large number of states. Since cancellation of requests and movement or destruction of links can happen at both ends simultaneously, the number of possible scenarios is large. The more elaborate cases, together with full details of the protocol, are discussed in a technical report [4].

### B. Absolutes and Hints

In order to facilitate efficient delivery of messages, Charlotte attempts to keep consistent, up-to-date information at both ends of each link. Link movement therefore requires that a third party (the kernel at the far end of the link that is moving) be informed. That third party may have a pending *Send* or *Receive* of its own. It may even be moving *its* end. The protocol is entirely symmetric; neither end of a link plays a dominant role. Although a link can be moved with very few kernel-level messages, the possible interleavings are subtle enough that we were forced to abandon a half dozen "final" algorithms before arriving at a correctness proof.

An alternative approach to link movement would rely on a system of hints. Each end of a link would keep track of the *probable* location of the other end, but the link movement protocol could leave this data inconsistent. Except in cases where a link moves more than once before being used to send a message (in which case hints are more efficient), the total numer of kernel-level messages would be the same with both approaches. A message sent to the wrong location would need to be returned, so the hint could be updated. We now believe, however, that hints would have permitted a substantially smaller kernel. We would be inclined to dispense with absolutes in future implementations.

### C. Interrelations

We designed our implementation so that all communication scenarios, both simple and complex, could be handled in a regular manner. The protocol is directed by a hand-built, table-driven finite state automaton. The states of the automaton reflect the status of a link. There are twenty types of input events, six of which represent requests from local processes and the rest of which represent messages from remote kernels. In an attempt to ensure correctness of the tables, we manually enumerated and simulated an exhaustive list of cases.

Our original hope was that the implementations of the various communication primitives would be more or less orthogonal. In practice, however, the interrelations between events on a link were surprisingly complex. An action at one end can occur when the other end is in an arbitrary state. Since *Send* and *Receive* are nonblocking, even the local end of a link may be in any of a large number of states when a related request is made.

As is true in other systems [12] the number of automation 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. *Cancel* is implemented in the *Send* and *Receive* automata. We also reduce the number of states with a method described by Danthine [16] and others [11] in which some information is encoded in global variables. The variables are consulted only in particularly complex cases.

Our automata have approximately 250 non-error entries, each of which has a prescribed action. Many actions apply 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.

### D. Kernel Structure

The kernel itself is implemented as a collection of nonpreemptable Modula processes [20], which we call *tasks* to distinguish them from user-level processes. These tasks communicate via queues of work requests. The Automaton Task implements all four automata. Requests from processes are first verified by the Envelope Task. Communication requests are then forwarded to the automaton's work queue. Two tasks manage 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. User processes run only when kernel tasks have nothing left to do.

The division of labor along functional lines made the kernel relatively easy to build. We have also found it easy to maintain. Most errors can be traced to an isolated section of code, and modifications rarely have widespread implications. We have, for example, implemented process migration as an incremental enhancement of Charlotte without substantially modifying the automata [6], [7]. On the other hand, the complete kernel/kernel protocol is almost beyond the comprehension of any single person. In this sense, the goal of simplicity has clearly not been met.

## IV. 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. Errors occur not only within individual processes, but also in the interactions between processes. *Ordering errors*, in particular, arise from unexpected interleavings of asynchronous events [23]. In addition to worrying about the more global ordering of messages received from different places (a subject beyond the scope of our work), processes in Charlotte must also be careful not to use a *Receive* buffer or modify a *Send* buffer before the associated kernel request has finished. In servers that manage large numbers of buffers, mistakes are more common than one might at first expect.

We found writing server processes to be suprisingly difficult. Ordering errors were not the only program. Several others can be attributed directly to our use of a conventional sequential language (a Modula subset), with ordinary kernel calls for interprocess communication. In particular:

- 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 very hard to find.
- Every kernel call returns a status value that indicates whether the requested operation succeeded or failed. Different sorts of failures result in different values. A well written program must inspect every status and be prepared to deal appropriately with every possible value. It is not unusual for 30 percent of a carefully written server to be devoted to error checking and handling. Even an ordinary client process must handle errors explicitly, if only to terminate when a problem occurs.
- 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. Charlotte servers therefore adopt an alternative program structure in which a single global loop surrounds a case statement that han-

dles arbitrary incoming messages. This explicit interleaving of separate conversations is very hard to read and understand.

Previous research had addressed these concerns in several different ways. The problem of message packing and unpacking has been solved in several distributed systems by the development of remote procedure call stub generators. Birrell and Nelson's Lupine [10] and the Accent Matchmaker [24] are particularly worthy of note. Safety depends on integrating the stub generator into the compiler's type-checking mechansim and on preventing messages from being sent in any other way. If the language provides facilities for exception handling, then the problem of checking result values can be solved with stubs as well.

Addressing the problem of conversation management requires multiple cooperating threads of control in a single address space. Such threads are supported directly by the Amoeba [28] and Mach [1] distributed operating systems and may be realized through programming conventions in any operating system that allows processes to share memory. There is, however, a nontrivial cost associated with scheduling a server's threads at the operating-system level, since creating a thread or switching from one thread to another requires a context switch into and out of the kernel. The designers of the Medusa distributed operating system [29] chose to implement coroutines at the user level rather than change the set of threads (activities) in a server (task force) at run time.

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, possibly including lightweight processes. In addition to the performance problems alluded to above, this approach assists only those applications for which the particular choice of abstractions is appropriate, and is likely to make it *more* difficult to write applications for which the abstractions are *not* appropriate. Our preference is to provide a higher-level interface *on top of* the communication kernel.

## V. The Lynx Programming Language

In keeping with the conclusions of the preceding section, we have developed a language called Lynx. It is described in full detail in several other places [32], [36], [37]. Lynx differs from most other distributed languages we have surveyed [34] in three major areas:

*Processes and Modules:* Processes and modules in Lynx reflect the structure of a multicomputer. Modules may nest, but ony within a node; no module can cross the boundaries between nodes. 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. In contrast to the lightweight processes of most distributed programming languages, the threads of Lynx are coroutines with no pretense of parallelism.

*Communication Paths and Naming:* Lynx provides Charlotte links as first-class language objects. The programmer has complete run-time control over the binding of the links to processes and the binding of names to links. The resulting flexibility allows the links to be used for reconfigurable, type-checked connections between very loosely coupled processes—processes designed in isolation and compiled and loaded at 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 approch(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, separate threads of control can be used to manage conversations with separate clients. 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 as a pointer accesses an object in Pascal. 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 run-time error results from any attempt to use them.

From a client's point of view, message passing looks like a remote procedure call; the sending thread of control transmits a request and waits for a reply.[1] An active thread can serve a request for a given operation by executing an Ada-like [40] *accept* statement. A process can also ar-

[1] Since links are completely symmetric, the terms "client" and "server" are relevant only in the context of a given call. It is entirely possible for the processes at opposite ends of a link to make requests of each other simultaneously. The file system, for example, may be a client of the memory manager when it needs more buffer space, while the memory manager may be a client of the file system when it needs to load some data into memory. One link between the processes suffices.

range to receive requests implicitly by binding a link to an entry procedure. *Bind* and *unbind* are executable commands. When all threads in a process are blocked, the run-time support package attempts 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 or 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 messages. A novel mechanism for self-descriptive messages [33] allows the checking to be performed efficiently at run time. An exception-handling mechanism permits recovery from errors that arise in the course of message passing and allows one thread to interrupt another.

Our experience indicates that it is far easier to write servers in Lynx than in a sequential language with calls to Charlotte primitives. Development time, source code length, and frequency of bugs are all reduced significantly. Message transmission time is increased by less than ten percent, even when Lynx code is compared to Modula programs that perform no error checking.

We have used Lynx to reimplement several of the server 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, taveling salesman) [18], [19], [21]. The Charlotte link concept, as represented in Lynx, has proven to be a valuable abstraction for representing resources and algorithm structure. This vindicates our original choice of bidirectional links.

On the other hand, the structure of the run-time support package for Lynx has led us to doubt the appropriateness of the interface between that package and the kernel. Despite the fact that much of the design of Lynx was motivated by the primitives of Charlotte, the actual implementation proved to be quite difficult. For example, Lynx requires greater selectivity than Charlotte provides for choosing an incoming message. There is no way to tell the kernel to accept reply messages but to ignore requests. In addition, Lynx permits an arbitrary number of links to be enclosed in a message, while Charlotte supports only one. Implementations of Lynx for SODA [25], a "Simplified Operating system for Distributed Applications," and the BBN Butterfly Parallel Processor [8] were in some ways considerably simpler [35].

## VI. Lessons

### A. The Kernel/Process Interface

- *Duplex links are a useful abstraction.* We have been generally happy with the success of Charlotte links. Messages can be sent in both directions without resorting to artificial "reply links." Symmetry means that processes need not keep track of which end of a link is which. Receivers have as much control as senders over the links on which they are willing to communicate. Processes at each end can be informed of important events (such as termination) at the other end. On the other hand, the protocol for link transfer would be much less complex if only one end could move. In addition, certain facilities not provided in Charlotte, such as multicast and broadcast, do not appear to be compatible semantically with the notion of point-to-point software connections. For processes that use a remote-procedure-call style of interaction, there is no obvious way to forward a request on a link.[2] The verdict, therefore, is mixed. There are clearly communication paradigms for which duplex links are not an attractive abstraction. For a very large class of problems, however, our experience suggests that links work very well.

- *Synchronous notifications work well with nonblocking primitives.* The combination of nonblocking *Send* and *Receive* with blocking *Wait* allows processes to communicate with large numbers of peers without unnecessary delays and without sacrificing the ability to obtain synchronous notification of errors. In the absence of state-sharing lightweight processes, we are unaware of any other mechanism that provides a comparable level of function. In retrospect, we believe that a system of unique identifiers for outstanding requests would be a useful enhancement to Charlotte.

- *Message screening belongs in the application layer.* Every reliable protocol needs top-level acknowledgments [31]. 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 process and the kernel and if the descriptive facilities in the kernel/process interface are sufficiently rich to specify precisely which messages are wanted. The descriptive facilities of Charlotte allow a user to specify nothing more than the name of a link. For the run-time package of Lynx, a finer degree of screening was desired. We would be tempted in future systems to allow multiple outstanding *Sends* and to adopt a mechanism similar to that of SODA [25], in which acknowledgments are delayed until the receiving process has examined the message and decided it really wants it.

- *Middle-level primitives are not a good idea.* A very low-level kernel/process interface is almost certain to be too cumbersome for programmers to use directly. It will, however, admit a wide variety of higher-level packages. A very high-level interface may be easy to use, but only for a single style of application program. In order to permit direct use by many kinds of application programs, the communication facilities of most distributed operating systems (Charlotte among them) have been designed with a middle-level interface. While this intermediate ap-

[2] In NIL [39], for example, one sends to a port, but replies to a message. There need not be an explicit path between the replier and the original requester.

proach may support both top-level applications and additional layers of software, our experience suggests that it fills neither role particularly well. Charlotte is a little too low-level for everyday use by ordinary programmers, and a little too high-level for the efficient implementation of certain parts of Lynx. The proliferation of remote-procedure-call stub generators for other distributed operating systems suggests that many researchers have arrived at similar conclusions.

### *B. Implementation Considerations*

- *Finite-state protocol machines are extremely useful.* Our rigidly structured (if handwritten) automaton provides a space- and time-efficient implementation of a very complex protocol. Construction of the automaton was straightforward, and the systematic enumeration of states provided us with a high degree of confidence in the correctness of our implementation. The division into subautomata and the judicious use of global flags were useful simplifying techniques. We recommend them highly.
- *Division of labor among tasks is elegant but slow.* By assigning different functions to different Modula tasks, we were able to subdivide the kernel into essentially independent pieces. The interfaces between pieces were simple queues of notices. The modularity of the kernel has made maintenance relatively easy, but has not been particularly good for performance. A simple message between machines takes approximately 25 ms, a substantial fraction of which is devoted to task switches in the sending and receiving kernels. We would be tempted in future projects to consider less expensive structuring techniques (such as upcalls [14], for example).
- *Absolute distributed information is hard to maintain.* Consistent, up-to-date, distributed information can be more trouble than it is worth. It may be easier to rely on a system of hints, so long as 1) they usually work, and 2) we can notice and recover when they fail. We suspect that the size of the Charlotte kernel could be reduced considerably by using hints for the location of link ends.

### *C. General Lessons for Parallel Systems*

- *Simple primitives may interact in complicated ways.* Even concise, correct semantics may require surprisingly complicated algorithms. At first glance, the primitives to make and cancel requests and to destroy and transfer links might appear to be largely orthogonal. When occurring simultaneously, however, these "simple" ideas become complex. In lieu of specific advice for avoiding interactions, we can at least suggest that future researchers distrust their native optimism.
- *Confidence in concepts requires an implementation.* It is difficult to anticipate all ramifications of an idea when exploring it on paper alone. Even with unlimited self-discipline, the lack of practical proof techniques means that building and testing an actual implementation is the best way to gain confidence in the validity of one's ideas. We do not believe that we were particularly naive in expecting our implementation to be significantly smaller than it is. It was only after we were deep into the enumeration of automaton states that the full nature of the problem became apparent.
- *Explicit management of concurrency is difficult.* The combination of nonblocking requests and the lack of kernel buffering makes it easy to overwrite a buffer. The presence of multiple outstanding requests means that more than one buffer must be used, and that appropriate context must be managed for each. The problems are particularly severe for server processes, which must interleave conversations with a large number of clients and which can never afford to wait for specific requests.
- *Appropriate high-level tools can mitigate programming problems.* Lynx was designed to a large extent in response to the previous lesson. Other tools were built to address a variety of other concerns. Library packages that understand how to talk to servers make it easier to write simple clients. A *connector* utility initializes multiprocess applications with arbitrary link connections as described by configuration files. The success of these tools suggests that, as on conventional uniprocessors, the friendliness of a programming environment is more a reflection of the quality of its tools than of its operating system primitives. The goal of the kernel designer should not be to support application programs so much as to support the higher-level systems software that in turn supports the applications.

## Acknowledgment

The authors would like to thank the entire Charlotte team not only for their assistance in designing the operating system, but also for contributing to earlier drafts of this paper. M. H. Solomon was a principal designer of Charlotte. Early versions of the kernel were implemented by P. Krueger and A. Michael. B. Rosenburg built early versions of the server processes, and had a major influence on the design of the interkernel protocol. C.-Q. Yang maintained the servers and converted them to Lynx. T. Virgilio and B. Gerber designed, implemented, and maintained the nugget software. We also profited from many discussions with P. Dewan, A. J. Gordon, W. K. Kalsow, J. Kepecs, and H. Madduri.

## 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," in *Proc. Summer 1986 USENIX Tech. Conf. Exhibition*, June 1986.

[2] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe, "The Eden system: A technical review," *IEEE Trans. Software Eng.*, vol. SE-11, no. 1, pp. 43-59, Jan. 1985.

[3] 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," *ACM TOPLAS*, vol. 10, no. 1, pp. 51-86, Jan. 1988.

[4] Y. Artsy, H.-Y. Chang, and R. Finkel, "Charlotte: Design and implementation of a distributed kernel," Dep. Comput. Sci., Univ. Wisconsin—Madison, Tech. Rep. 554, Aug. 1984.

[5] ——, "Interprocess communication in Charlotte," *IEEE Software*, vol. 4, no. 1, pp. 22-28, Jan. 1987.

[6] ——, "Processes migrate in Charlotte," Dep. Comput. Sci., Univ. Wisconsin—Madison, Tech. Rep. 655, Aug. 1986.

[7] Y. Artsy and R. Finkel, "Simplicity, efficiency, and functionality in designing a process migration facility," in *Proc. Second Israel Conf. Computer Systems and Software Engineering*, IEEE, May 1987.

[8] BBN Laboratories, "Butterfly® parallel processor overview," Rep. 6149, Version 2, Cambridge, MA, June 16, 1986.

[9] F. Baskett, J. H. Howard, and J. T. Montague, "Task communication in Demos," in *Proc. Sixth ACM Symp. Operating Systems Principles*, Nov. 1977, pp. 23-31.

[10] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39-59, Feb. 1984; originally presented at the Ninth ACM Symp. Operating Systems Principles, Oct. 10-13, 1983.

[11] G. V. Bochmann, "Finite state description of communication protocol," *Comput. Networks*, vol. 2, pp. 361-372, 1978.

[12] G. V. Bochmann and J. Gescei, "A unified method for the specification and verification of protocols," *Inform. Processing*, IFIP, 1977.

[13] D. Cheriton, "The V kernel—A software base for distributed systems," *IEEE Software*, vol. 1, no. 2, pp. 19-42, Apr. 1984.

[14] D. Clark, "The structuring of systems using upcalls," in *Proc. Tenth ACM Symp. Operating Systems Principles*, Dec. 1-4, 1985, pp. 171-180; in *ACM Operat. Syst. Rev.*, vol. 19, no. 5.

[15] R. Cook, R. Finkel, D. De Witt, L. Landweber, T. Virgilio, "The Crystal nugget: Part I of the first report on the Crystal project," Dep. Comput. Sci., Univ. Wisconsin—Madison, Tech. Rep. 499, Apr. 1983.

[16] A. Danthine and J. Bremer, "An axiomatic description of the transport protocol of cyclades," in *Proc. Professional Conf. Computer Networks and Teleprocessing*, Mar. 1976.

[17] D. J. DeWitt, R. Finkel, and M. Solomon, "The Crystal multicomputer: Design and implementation experience," *IEEE Trans. Software Eng.*, vol. SE-13, no. 8, pp. 953-966, Aug. 1987.

[18] R. Finkel, 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," Dep. Comput. Sci., Univ. Wisconsin—Madison, Tech. Rep. 630, Feb. 1986.

[19] R. Finkel, B. Barzideh, C. W. Bhide, M.-O. Lam, D. Nelson, R. Polisetty, S. Rajaraman, I. Steinberg, and G. A. Venkatesh, "Experience with Crystal, Charlotte, and LYNX: Second report," Dep. Comput. Sci., Univ. Wisconsin—Madison, Tech. Rep. 649, July 1986.

[20] R. Finkel, R. Cook, D. DeWitt, N. Hall, and L. Landweber, "Wisconsin Modula: Part III of the first report on the Crystal project," Dep. Comput. Sci., Univ. Wisconsin—Madison, Tech. Rep. 501, Apr. 1983.

[21] R. Finkel, G. Das, D. Ghoshal, K. Gupta, G. Jayaraman, M. Kacker, J. Kohli, V. Mani, A. Raghavan, M. Tsang, and S. Vajapeyam, "Experience with Crystal, Charlotte, and LYNX: Third report," Dep. Comput. Sci., Univ. Wisconsin—Madison, Tech. Rep. 673, Nov. 1986.

[22] R. Finkel, M. Solomon, D. DeWitt, and L. Landweber, "The Charlotte distributed operating system: Part IV of the first report on the Crystal project," Dep. Comput. Sci., Univ. Wisconsin—Madison, Tech. Rep. 502, Oct. 1983.

[23] A. J. Gordon, "Ordering errors in distributed programs," Ph.D. dissertation, Dept. Comput. Sci., Univ. Wisconsin—Madison, Tech. Rep. 611, Aug. 1985.

[24] M. B. Jones, R. F. Rashid, and M. R. Thompson, "Matchmaker: An interface specification language for distributed processing," in *Conf. Rec. Twelfth Annu. ACM Symp. Principles of Programming Languages*, Jan. 1985, pp. 225-235.

[25] J. Kepecs and M. Solomon, "SODA: A simplified operating system for distributed applications," *ACM Operat. Syst. Rev.*, vol. 19, no. 4, pp. 45-56, Oct. 1985; originally presented at the *Third ACM SIGACT/SIGOPS Symp. Principles of Distributed Computing*, Aug. 27-29, 1984.

[26] B. Liskov and R. Scheifler, "Guardians and actions: Linguistic support for robust, distributed programs," *ACM TOPLAS*, vol. 5, no. 3, pp. 381-404, July 1983.

[27] B. P. Miller, D. L. Presotto, and M. L. Powell, "DEMOS/MP: The development of a distributed operating system," *Software—Practice and Exp.*, vol. 17, pp. 277-290, Apr. 1987.

[28] S. J. Mullender and A. S. Tanenbaum, "The design of capability-based distributed operating system," *Comput. J.*, vol. 29, no. 4, pp. 289-299, 1986.

[29] J. D. Ousterhout, D. A. Scelza, and S. S. Pradeep, "Medusa: An experiment in distributed operating system structure," *Commun. ACM*, vol. 23, no. 2, pp. 92-104, Feb. 1980.

[30] R. F. Rashid and G. G. Robertson, "Accent: A communication oriented network operating system kernel," in *Proc. Eighth ACM Symp. Operating Systems Principles*, Dec. 14-16, 1981, pp. 64-75; in *ACM Oper. Syst. Rev.*, vol. 15, no. 5.

[31] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 277-288, Nov. 1984.

[32] M. L. Scott, "Design and Implementation of a distributed systems language," Ph.D. dissertation, Dept. Comput. Sci., Univ. Wisconsin—Madison, Tech. Rep. 596, May 1985.

[33] M. L. Scott and R. A. Finkel, "A simple mechanism for type security across compilation units," *IEEE Trans. Software Eng.*, vol. 14, no. 8, pp. 1238-1239, Aug. 1988.

[34] M. L. Scott, "A framework for the evaluation of high-level languages for distributed computing," Dep. Comput. Sci., Univ. Wisconsin—Madison, Tech. Rep. 563, Oct. 1984.

[35] ——, "The interface between distributed operating system and high-level programming language," in *Proc. 1986 Int. Conf. Parallel Processing*, Aug. 19-22, 1986, pp. 242-249.

[36] ——, "LYNX reference manual," Dep. Comput. Sci., Univ. Rochester, BPR 7 (revised), Aug. 1986.

[37] ——, "Language support for loosely-coupled distributed programs," *IEEE Trans. Software Eng.*, vol. SE-13, pp. 88-103, Jan. 1987.

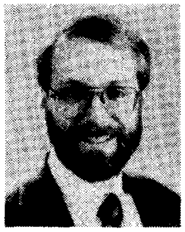
[38] M. H. Solomon and R. A. Finkel, "The Roscoe distributed operating system," in *Proc. Seventh ACM Symp. Operating Systems Principles*, Dec. 1979, pp. 108-114.

[39] R. E. Strom and S. Yemini, "NIL: An integrated language and system for distributed programming," *Proc. SIGPLAN '83 Symp. Programming Language Issues in Software Systems*, June 27-29, 1983, pp. 73-82; in *ACM SIGPLAN Notices*, vol. 18, no. 6.

[40] United States Dep. Defense, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983, Feb. 17, 1983.

**Raphael A. Finkel** was born in 1951 in Chicago, IL, where he attended the University of Chicago, receiving the Bachelor's degree in mathematics and the Master of Arts degree in teaching. He received the Ph.D. degree from Stanford University, Stanford, CA, in 1976 in the area of robotics.

From 1976 to 1987, he was a faculty member of the University of Wisconsin—Madison. He has been a Professor of Computer Science at the University of Kentucky in Lexington since 1987. His research involves distributed data structures, interconnection networks, distributed algorithms, and distributed operating systems. He has received several teaching awards and has published an introductory text on operating systems.



**Michael L. Scott** (S'85–M'85) is a graduate of the University of Wisconsin—Madison, where he received the B.A. degree in mathematics and computer sciences in 1980, and the M.S. and Ph.D. degrees in computer sciences in 1982 and 1985, respectively.

He is now an Assistant Professor in the Department of Computer Science at the University of Rochester. He is the recipient of a 1986 IBM Faculty Development Award. His research focuses on programming languages, operating systems, and program development tools for parallel and distributed computing. He is co-leader of Rochester's Psyche project, which centers on the design and implementation of an ambitious new style of operating system for large-scale shared-memory multiprocessors.

**Yeshayahu Artsy** received the B.A. degree in political sciences and statistics from the Hebrew University of Jerusalem in 1975, the B.A. degree in economics and the M.B.A. degree in management information systems from Tel Aviv University in 1979 and 1981, respectively, and the M.S. and Ph.D. degrees in computer sciences from the University of Wisconsin—Madison in 1984 and 1987, respectively.

He has been with the Distributed Systems Advanced Development Group at Digital Equipment Corporation since 1987. Previously, he managed the Tel Aviv University Management School Computing Center (1976–1979), and was in charge of system software support for Burroughs Medium Systems in Israel (1979–1982). His interests include distributed operating systems and services, open systems, and object-oriented programming.

**Hung-Yang Chang** received the B.S.E.E. degree from National Taiwan University in 1979, and the M.S. and Ph.D. degrees from the University of Wisconsin—Madison in 1983 and 1987, respectively. His dissertation is a study of distributed soft real-time scheduling algorithms for future complex real-time systems.

He joined the IBM Thomas J. Watson Research Center in 1987, where he is a research staff member of the department of Parallel System Architecture and Software. He is currently working on job scheduling and performance measurement tools for a large-scale shared-memory multiprocessor. His research interests include task scheduling algorithms, performance measurement of parallel software, and object-oriented multiprocessor kernel design.