INTERPROCESS COMMUNICATION IN CHARLOTTE

by

Yeshayahu Artsy
Hung-Yang Chang
Raphael Finkel

Computer Sciences Technical Report #632

February 1986

Interprocess Communication in Charlotte [1]

Yeshayahu Artsy
Hung-Yang Chang
Raphael Finkel

University of Wisconsin — Madison

Computer Sciences Department

February 1986

Abstract

Charlotte is a distributed operating system that provides a powerful interprocess communication mechanism. This mechanism differs from most others in that it employs duplex links, does not buffer messages in the kernel, and does not block on communication requests. A *link* is a bound communication channel between two processes upon which messages can be sent, received, awaited, or canceled. Processes may acquire new links, destroy their end of the link, or enclose their end as part of an outgoing message. A link is thus a dynamic capability-like entity that permits only the holder access to its communication resource.

We first introduce the Charlotte interprocess communication structure and discuss our motivation for choosing the specific operations. We then describe distributed computing environments in which Charlotte is applicable. Next, we discuss how we implemented the interprocess-communication facility. We close with a comparison between Charlotte and related research and justify our design and its implementation cost.

[1]This work was supported in part by NSF grant MCS-8105904 and by DARPA contracts N00014-82-C-2087 and N00014-85-K-0788.

CONTENTS

1. Introduction

Charlotte is a distributed operating system that provides a powerful interprocess communication mechanism employing duplex *links* to facilitate distributed applications. A *link* is a bound communication channel between two processes upon which messages can be sent, received, awaited, or canceled. Processes may acquire new links, destroy their end of the link, or enclose their end as part of an outgoing message. A *link* is thus a dynamic capability-like entity that permits only the holder access to its communication resource.

The idea of using links for interprocess communication was introduced in the Demos operating system [1]. Charlotte has enhanced the idea in several ways. First, a Charlotte link is a duplex communication channel, while a Demos link allows traffic in only one direction. Both holders of a Charlotte link have equal rights to use, transfer, or destroy it. Moreover, they can do so simultaneously, independently from each other. This symmetry permits a uniform communication interface. Second , Charlotte does not buffer messages between processes. When both processes are in the same machine, Charlotte copies the data directly between user address spaces. This decision places the burden of buffer management on the programmer, not the kernel. Kernel buffering often requires an extra kernel copy, which can be a waste of both space and time. Our experience with Arachne, a predecessor of Charlotte, shows that buffer management at the kernel level leads to complex deadlock [2]. Third, neither the *Send* nor the *Receive* primitives block the caller. This design allows servers to proceed independently of the speed of their clients.

We first introduce the Charlotte interprocess communication structure and discuss our motivation for choosing the specific operations. We then describe distributed computing environments in which Charlotte is applicable. Next, we discuss how we implemented the interprocess-communication facility. We close with a comparison between Charlotte and related research and justify our design and its implementation cost.

2. Charlotte interprocess communication

The design space of message-based interprocess communication mechanisms is large. The issues of naming, addressing, and flow control intertwine with one another. There is no general agreement on how the kernel should handle these issues. At one extreme, the kernel could present the data-link layer of a local area network, allowing users to tailor their protocols to the needs of particular programs. Each process would specify destination machines and process numbers. The Rochester Intelligent Gateway [3] has such a simple interface, in which processes communicate through an intermediary called a *port*. At the other extreme, user processes could transfer messages among themselves in a network-transparent and process-name-transparent manner. This approach is taken by Accent [4] and Demos/MP [5] as well as Charlotte. Charlotte tries to provide its users with an especially high-level facility.

2.1. Design decisions

The distinctive features of Charlotte are duplex communication, lack of buffering, nonblocking send and receive, synchronous wait for communication completion, selectivity of receipt, and dynamic link transfer. We discuss each feature in turn.

Duplex communication

Charlotte links are duplex. Simplex links have the following shortcomings.

- *Dangling links.* When a process dies, all the simplex links that point to it automatically become invalid. These links are scattered all over the network, and may also be in messages still in transit. It is impractical to search the entire system for links that may point to a particular process. Instead of revoking all these links at once, Demos/MP invalidates them when they are used. In the meantime, they tie up system space. The worst case occurs when the dead process is a server, since servers typically own many links.
- *Link-management overhead.* Client-server, master-slave, and remote-procedure-call situations all require information to flow in both directions. Even pipelines require reverse flow for

exception reporting. With simplex links, reverse flow requires reply links. Experience with Arachne indicates that many short-lived reply links are created, used once, and discarded during remote requests.

- *Migration.* Demos/MP leaves a *stub* behind whenever a process migrates. The alternative is to broadcast the new destination of the moving process from which all kernels correct their associated links. Duplex links do not require such mechanisms.

Charlotte attempts to avoid these problems by providing duplex links. This choice introduces new problems and complexities, as we shall see.

Buffering

An infinite number of intermediate buffers offers 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.

Charlotte experiments with the other extreme, avoiding buffering altogether. With no intermediate system buffers, the kernel is simpler and more robust. Moreover, the user can assert complete control over the number of messages in transit.

For applications requiring the transfer of large amounts of data, Charlotte eliminates the constraint of finite message size. Since buffers are allocated out of user address space, large transfers present no difficulties for the kernel implementation. However, Charlotte does provide a small cache of buffers purely for efficiency. The cache stores arriving messages that are not yet requested by the destination.

Blocking

We decided to experiment with semantics in which basic communication activities do not block. The *Send* and *Receive* service calls initiate communication but do not wait for its completion. In this way, a process may post *Send* or *Receive* requests on many links without waiting for any to finish. This design allows servers to communicate with clients without fear of being blocked. (Blocking forms of *Send* and *Receive* are also available.)

Waiting

Posting a *Send* or *Receive* is synchronous, but completion is inherently asynchronous. We provide several facilities for dealing with this asynchrony. First, a process may explicitly wait for a *Send* or *Receive* to finish. Second, a process may poll the completion status of a *Send* or *Receive*. Third, a process may convert the completion event into an interrupt. (We have deferred implementing this third facility due to semantic clashes with other features.)

Selectivity of receipt

The *Receive* request can specify a single link or any link. An intermediate specification might be useful, but it imposes an extra burden on the kernel. Similarly, the *Wait* request can specify a single link or all links. It can also specify whether a send or receive event (or either) is to be awaited.

Link moving

The ownership of an end of a link is transferable. A process may enclose a link in a message sent across a different link. The recipient of that message becomes the new owner of the link. The sending process loses its ownership. While a link is moving, the process at the other end is still able to post *Send* or *Receive* requests or even move its end of the link. The entire operation of link moving is transparent to the owner of the other end of the link.

2.2. Communication primitives

Charlotte provides five major operations on links. Their descriptions are detailed elsewhere [6, 7]. We summarize their main features here.

Send(transmission link, buffer, buffer size, enclosed link)

initiates a transfer of data from the indicated buffer along the transmission link. The operation remains in progress until its completion is reported by the *Wait* call. An end of a link may be enclosed in the message. If *Send* should fail, the enclosed link, if any, is restored to the sender.

Receive(transmission link, buffer, buffer size)

allows a message to be received on the indicated link and placed in the buffer. The link identifier *AllLinks* permits the acceptance of a message on any link held by that process. *Receive* on *AllLinks* may not coexist with *Receive* on a specific link, because it may cause inconsistency (from user point of view) in choosing the buffer for arriving messages. If the buffer is not large enough to hold the entire message, as much as fits is placed in the buffer, and the tail is lost. In any case, both the sender and the receiver are notified via the completion event (discussed shortly) how much data was accepted.

Wait(transmission link, direction): completion event

returns the result of a previous communication request. The direction may be incoming (*Receive*), outgoing (*Send*), or both, on a specific link or any link. An event descriptor is returned by the kernel. It contains the matched link, direction, completion code, number of bytes transmitted, and a new link, if one was acquired. The user may choose to be blocked until the operation completes or to continue immediately.

Cancel(transmission link, direction)

requests cancellation of a *Send* or a *Receive* operation on the specified link. *Cancel* returns an error when the operation does not exist, and failure when the operation has progressed beyond the point where the kernel can stop it. This call blocks the process until the kernel is able to report success or failure.

Destroy(link)

Requests that the given link be closed. *Destroy* always succeeds unless the link does not exist or is being transferred. This call will abort all outstanding *Send* or *Receive* requests on that link. It blocks the user until any necessary cooperation by the other kernel has completed. The process at the remote end will get a failure code from any uncompleted call related to that link. That end is reclaimed once a remote process invokes *Destroy* on its end. A request to destroy a link being transferred is an error. All links owned by a process are destroyed when it terminates.

3. Using Charlotte links

We have had extensive experience in using the Charlotte interprocess communication semantics, and have found it quite appropriate to our needs. We have implemented standard utilities, such as a fileserver, a name server, a memory manager, a link connector, and a command interpreter [7], all of which use links to represent resources and services. More recently, the Lynx programming language, whose communication semantics are based on Charlotte links, has been implemented and is in heavy use [8]. We have converted several of our utilities from Modula to Lynx. The new versions are easier to code and maintain because they package the link concept in a type-secure, multithread environment. We have also used Lynx to encode several applications such as B trees, systolic arrays, incremental minimal spanning trees, and Prolog and-or tree search [9].

Charlotte's link facility fits numerous distributed-communication patterns. These patterns include a community of servers, remote procedure call, peer conversations, and interposed processes. In the following discussion, we show how to employ Charlotte interprocess-communication mechanisms in those applications.

- **Community of servers.**

Resource allocation in a distributed environment may be governed by a community of server processes for each resource. Ideally, the organization within the community of servers is hidden from the client. This server-client relationship can be implemented by a link between the client and any server in the community. If a particular server cannot satisfy a client's request, it can relay the request to another member of the community. Relaying involves passing the server's end of the link to another server. Link passing also allows servers to redistribute their load. The user is oblivious to the change of servers. Similarly, the client can transfer its end of the link to another process without interfering with the server community.

The non-blocking nature of *Send* and *Receive* allows servers to send requests to each other without waiting for the responses. The community of Starter processes in Charlotte fits this model well. A client may speak to any member of this community when it wants to start a new process. The individual Starters communicate with each other to share load statistics and to forward client requests. Furthermore, read-ahead and write-behind are easy to implement with non-blocking *Send* and *Receive* requests.

- **Remote procedure call.**

Remote procedure call [10] is the synchronous language-level transfer of control between programs in disjoint address spaces. It requires two-way communication. A first message holds input arguments, and a second contains the reply. The sender of the first message is blocked until the reply is received. Links support this activity naturally. Since Charlotte does not restrict the size of messages, no special effort is needed to deal with large parameters. Other communication systems [11, 12, 13] restrict message size.

In addition, the non-blocking nature of Charlotte communication allows several remote procedure calls to be active at one time. Similarly, the callee can accept calls in whatever order it prefers, reply more than once, or not reply at all. The callee can transform the call into another remote procedure call. The second call can return directly to the original caller.

We have used these features to implement a distributed B-tree application [9]. Different subtrees are under the control of different processes. A query starts at the root process and is directed by means of a remote procedure call to the appropriate subtree. While the result is pending, a second query may be submitted to the root and converted into another remote procedure call.

- **Conversation.**

Unlike remote procedure call, a *conversation* has two processes engage in a frequent exchange of information without the notion of master and slave. Either process can initiate a communication at any time. Such a conversational partnership can be represented by a single link. Other communication schemes require more channels [4, 5].

Charlotte implements file service in this way. An open file is represented by a link between the client and the server. At first, the link is used in a remote-procedure manner. The client can direct the file server to can enter a *stream protocol*. Then, if the file is open for reading, the server sends a continual stream of information to the client. If it is open for writing, information flows the other way. The reverse direction is still available for out-of-band information, such as requests to seek (for reading) and error reports (for writing).

- **Interposed processes.**

An interposed process can help two processes communicating either using different protocols or at different speeds. In the former case, the interposed process translates the protocols in both directions. In the latter case, the interposed process buffers the data from the producer until needed by the consumer. Other uses of interposed process include debugging and redirection of I/O. A debugging process may be arranged to monitor all interprocess communication by interposing itself on all links [14]. In all cases, the notion of a link provides a convenient vehicle to connect processes through an intermediary. However, interposing an intermediary can distort the Send/Receive semantics of the communicating processes.

4. Implementation

Charlotte is implemented on the Crystal multicomputer [15]. It resides above a communication package called the *nugget* [11], which provides a reliable transmission service. Charlotte's kernel implements the abstractions of processes and links. In order to provide the facilities described earlier, kernels communicate with each other through a low-level communication protocol. Significant events for this protocol include messages received from remote kernels and requests from local processes.

4.1. The complexity of the protocol

A protocol design for a set of communication primitives must incur a complexity penalty proportional to the complexity of the primitives. However, commonly used primitives should be executed efficiently. We have achieved this goal by separating common from unusual situations.

The easiest situation is when a *Send* includes no links, 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 a completion packet. If the matching *Receive* is posted later, but the message is still held in the cache on the receiving kernel, the same protocol applies. Extra complexity is introduced by messages too large to fit in a single packet.[2]

It is simple for the protocol to handle the *Wait* request. The variations that we allow (such as waiting for any outstanding *Receive*) do not add much complexity.

Cancellation of *Send* and *Receive* is slightly more complex, since it must deal with numerous cases. The *Send* or *Receive* in question might be in one of several states, such as still pending and matched. Further, both ends may simultaneously cancel an operation that has matched.

The greatest complexity is introduced by link movement and destruction, especially when both occur at the same time. Link movement requires that a third party (the kernel of the other end of the link that is moving) be informed. That third party may also have a *Send* or *Receive* pending. The protocol must also deal with both ends moving simultaneously.

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, or if the remote end is itself in motion, the situation becomes complex indeed. Such cases do occur, especially when a process terminates unexpectedly. We discuss these complex scenarios and present the protocol elsewhere [7]. The protocol we developed uses twenty message types, six of which represent process requests.

4.2. Implementation by finite state automata

Given the number of cases, some simple and some complex, we designed the implementation so that all cases are handled in a regular manner. Complex cases require more work, but they do

[2] The communication device imposes a limit of about 2KB per packet.

not require a special coding style.

Our prototype implementation of Charlotte used *ad hoc* coding methods. We found this style very difficult to debug and maintain. The current implementation represents the protocol as a finite state automaton. Each link end has a state, and the inputs to the automaton are process requests and packets arriving from remote kernels. The number of states is quite large, a phenomenon observed by others as well [16]. We reduced the number of states by building independent automata for different functions: Send, Receive, Destroy, and Move. These automata interact only in a few cases. For example, destroying a link affects its Send state. Cancellation is implemented in both the Send and Receive automata.

A second reduction is based on a method described by Danthine [17] and others [18] in which some state information is encoded in global flags. These flags are only consulted for particularly complex cases.

Our automata have about 250 non-error entries. About 100 different actions are performed. The simplest actions consist of a single operation, such as sending an acknowledgment packet. The most complex action checks about five flags and selects one of several operations.

4.3. The structure of the kernel

The kernel is implemented as a collection of Modula processes [19], which we call *tasks* to distinguish them from user and utility processes. These tasks communicate via queues of work requests. The *Automaton* task handles the protocol. The four automata are implemented as procedures within this task. 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 clock maintenance, statistics collection, and checking if other nodes are alive. Processes run only when kernel tasks have no work to do.

This division of labor simplifies the implementation. We have found the kernel relatively easy to debug and maintain. Most errors can be isolated to a specific action. Modifications are usually isolated to just a few actions.

5. Related work

In this section we compare our work with a representative sample of similar work done elsewhere. This comparison is based on communication models and implementations.

Demos/MP [5] is an extension of the Demos operating system [1] that operates in a distributed environment. Messages are sent on *links*. Links are buffered, one-way message channels to processes. In contrast, Charlotte links are unbuffered, two-way message channels. The successful implementation of process migration in Demos/MP has indicated that process migration is achievable in a link-based environment [5]. Migration in Charlotte is simpler in one sense, since there is no need to retain any information about the process in its old host. On the other hand, it is more complex in that Charlotte links contain more state information that must be moved.

Accent [4] is a descendent of the Rochester Intelligent Gateway [3]. Communication takes place through ports. A *port* is a protected kernel object into which messages may be placed and from which messages may be removed. Processes may have ownership, send, and receive rights on a port. Ownership and receive rights are not sharable. However, they may be given away. Therefore, a port can change its receiver, which is not possible for Demos links.

Both the Demos link and the Accent port provide only one-way communication. However, Accent does have reverse communication to warn senders when ownership or receive rights change on a port. Neither provides a Cancel facility. Demos and Accent provide an intermediate level of message selectivity, whereas Charlotte allows Receive to be posted on a single link or on all links. Unlike Charlotte, Accent provides a way for a receiver to preview an incoming message by inspecting a small amount of its information. This information can be used to decide which port to receive from. Soda [13] also allows the receiver to reject messages based on preview.

V-kernel [11] uses a global flat name space for specifying destinations, in contrast to the capability-like naming in Demos, Accent, and Charlotte. A global name space prevents network and process-name transparency. It also interferes with process migration and allows unwanted communication. By restricting its inter-process communication methods to support remote procedure call alone, V-kernel achieves a very efficient implementation of message passing. Charlotte, on the other hand, provides network and process-name transparency at the cost of transmission delay.

Eden [20, 21] is an object-based distributed system that uses remote invocation as the principal communication mechanism. This level of abstraction is higher than that provided by Charlotte, both in the fact that objects are supported (instead of processes) and that remote-invocation send is supported (instead of simple messages). In order to achieve this level, Eden employs a high-level language (EPL) that translates remote invocations into messages. Eden objects communicate with the Eden kernel through Accent; Eden kernels communicate with each other through Berkeley Unix* interprocess communication. Eden's use of EPL is similar to Charlotte's use of Lynx.

6. Conclusions

Charlotte is intended as a vehicle for large-scale distributed computation. Distributed applications can be programmed for Charlotte in several languages. The simplest interface is to use the communication primitives directly from a program written in C or Modula. A higher-level interface is provided by the Lynx language, which provides links as a first class object, provides strong typing on messages, and allows several threads of control within one process. We foresee special libraries for particular applications like backtracking [22] that completely hide the distributed nature of the computation.

Charlotte's link-based interprocess communication is a simple yet powerful tool. Once a link is established, it allows two processes to communicate. During that time, they may also engage in other activities, including communication activities. Like capabilities, links can be given away. This facility allows us to build natural structures for a number of communication patterns. A server can refer requests to other servers, and communities of servers can balance their load.

* Unix is a trademark of Bell Laboratories.

The ability to cancel arises from separating starting a communication event (*Send* or *Receive*) and waiting for it to complete. Only Charlotte and Soda [13] have this feature. We have found cancellation important in pipeline protocols. Later stages of the pipeline can send control requests to earlier stages. Those earlier stages cancel their current work and service the control request. Cancellation is also important in algorithms that use eager evaluation; a process can stop waiting for an answer if it is sure the answer is not needed.

The simple primitives that Charlotte provides interact in complex ways. To control the complexity, the implementation structures the protocol by using automata. The internal structure of the kernel is divided into independent tasks that communicate by work queues. The structure imposes an efficiency cost, but permits easy debugging and maintenance.

A legitimate question arising from our implementation is whether the facilities that Charlotte provides are worth the complexity of the protocol and the efficiency cost. One way to answer this question is to estimate the savings that would result from restricting or eliminating features. For example, we could restrict link motion to cases when the other end is idle. This restriction simplifies the protocol considerably.

We believe that it is too early to decide efficiency/feature tradeoffs. The field of distributed computation is too young to know average computing needs and behavior patterns. The uses to which Charlotte is put will shed light on this subject. We can say that the fairly heavy-duty links that Charlotte provides do introduce processing costs. On VAX-11/750 computers connected by an 80 Mbps token ring, a single send-receive pair takes about 25 ms, and the largest aggregate throughput for large messages is about 115 Kbps. To some extent, these figures are influenced by the nature of the token ring (it requires several interrupts on each end to send a message) and by the internal structure of the Charlotte kernel (we opted for modularity, leading to several task switches for each message sent, and we spend significant time in debugging-related functions).

The kernel of Charlotte and all utility processes are fully operational. Debugging support exists [23] and process migration has been implemented.

7. Acknowledgements

The Charlotte project owes its success to a great number of designers, implementers, and creative critics. Marvin Solomon together with Raphael Finkel are the principal designers of the entire Charlotte operating system. Bill Kalsow suggested the current structure of the kernel. Phil Krueger and Alan Michael implemented the prototype Charlotte kernel. Bryan Rosenburg was instrumental in designing early versions of the communication protocol. Many helpful ideas and comments were provided by other Charlotte group members: Prasun Dewan, Aaron Gordon, Vinod Kumar, Mike Litzkow, Hari Madduri, Michael Scott and Cui-Qing Yang.

8. References

1. F. Baskett, 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).

2. R. A. Finkel and M. H. Solomon, "The Arachne Distributed Operating System," Technical Report 439, University of Wisconsin–Madison Computer Sciences (July 1981).

3. K. A. Lantz, K. D. Gradischnig, J. A. Feldman, and R. F. Rashid, "Rochester's Intelligent Gateway," *Computer,* pp. 54-68 (October 1982).

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

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

6. R. Finkel, 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).

7. Y. Artsy, H-Y Chang, and R. Finkel, ``Charlotte: design and implementation of a distributed kernel,'' Computer Sciences Technical Report #554, University of Wisconsin–Madison (August 1984).

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

9. R. A. 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,'' Computer Sciences Technical Report #630, University of Wisconsin–Madison (February 1986).

10. B. J. Nelson, ``Remote procedure call,'' Technical report CMU-CS-81-119, Carnegie Mellon University (1981). PhD Thesis

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

12. D. R. Cheriton, M. A. Malcolm, L. S. Melen, and G. R. Sager, ``Thoth, a portable real-time operating system,'' *CACM* 22(2) pp. 105-115 (February 1979).

13. J. H. Kepecs 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).

14. R. D. Schiffenbauer, ``Interactive debugging in a distributed computational environment,'' Report LCS/TR-264, MIT Computer Science Department (September 1981).

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

16. G. V. Bochmann 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).

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

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

19. R. Finkel, 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).

20. E. Lazowska, H. Levy, G. Almes, M. Fischer, R. Fowler, and S. Vestal, ''The Architecture of the Eden System,'' *Proceedings of the Eighth Symposium on Operating Systems Priciples,* (December 1981).

21. G. T. Almes, 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).

22. R. Finkel and U. Manber, ''DIB — A distributed implementation of backtracking,'' *Proceedings of the Fifth International Conference on Distributed Computing Systems,* pp. 446-452 (May 1985).

23. A. J. Gordon and R. A. Finkel, ''The use of timing graphs for distributed program debugging,'' *Distributed processing technical newsletter,* (1984).