PROTECTING OBJECTS THROUGH THE USE OF PORTS

Stephen Vinter

Krithi Ramamritham

David Stemple

COINS Techinal Report 82-23

Department of Computer and Information Science
University of Massachusets, Amherst
Amherst, Massachusetts 01003

Abstract

In several systems, ports have been utilized as communication channels between cooperating processes. We extend the use of ports for achieving protection of shared objects. For this purpose, a port is viewed as an abstract data type and protection is achieved by restricting the operations that are available to processes for manipulating ports. Further, we equate ownership of a port with possession of a capability to request operations on an object. In this paper, we examine the design issues involved in such a paradigm of communication and protection. In particular, we investigate primitive port operations, port connection establishment, synchronization and cooperation among processes mediated by ports, and message structure. This investigation reveals those questions that need to be answered during the design of systems which use ports to achieve communication and protection.

1. Introduction

The primary purposes of an interprocess communication facility are process synchronization and interprocess data transfer. Both goals are achieved by the transfer of messages between processes. Data sharing is an efficient alternative to message transfer in a computer system, but is difficult to achieve in a distributed computing environment. This paper addresses port-based interprocess communication mechanisms in the context of distributed systems. In particular, it presents a taxonomy of various aspects of port-based communication, including addressing, connection topologies, primitive operations on ports, and message buffering techniques. After a brief survey of current protection mechanisms, the paper concludes by proposing ports as a means to implement capabilities. The thrust of this paper is to investigate the use of ports for protecting shared objects in an object-oriented system.

Port-based communication has a number of characteristics that distinguish it from other communication models. A port constitutes a communication path between sets of communicating, cooperating processes. In connecting communicating processes, a port can mask the identity of the processes involved in the communication. This feature, termed functional addressing, distinguishes port-based communication from communication mechanisms using other forms of addressing. The Connection Establishment section of this paper discusses attributes of communication connections that must be considered when the paths are formed, including addressing, restrictions placed on paths, and alternative topologies for communication connections.

Messages are placed in ports by a process with send access to the port. Messages are removed from ports by a process with receive access to the port. The section on Synchronization and Concurrency investigates alternative primitive operations on ports for placing messages in and removing messages from ports.

Ports provide a message buffering capability, queueing messages when communications are asynchronous. The Message Delivery section investigates topics related to the queueing of messages on ports. Issues discussed include message buffering, message prioritization and categorization, and aging. The section on Message Structure discusses the typing of data passed through ports.

A port is recognized by a system as an entity that is independent of the processes which use it. Ports are abstract data types, and hence, can only be manipulated by a predefined set of operations. In object-oriented systems, all data is represented by a set of objects, each object representing an instance of an abstract data type. Also, associated with each object is a manager process which controls the execution of operations on the object. Here we unite the notions of port-based communication, object-based systems and protection of objects by proposing ports not only for communication purposes but also as a mechanism for protecting objects. The capability of a user process to access a port that is connected to the manager of an object signifies the right to perform a particular set of operations on that object. Thus, ports are presented as an implementation of the concept of capabilities. Port-based protection is the subject of the final section.

2. Connection Establishment

The examination of the establishment of a communication connection requires investigating three issues: the addressing of processes involved in the communication connection, the restrictions imposed on the use of the communication path, and the topology of the communication path.

2.1 Addressing

Addressing denotes the method that a process uses to select other processes with which it desires to communicate. It is possible to address processes in a number of ways. We have chosen to categorize addressing as implicit, explicit, indirect, or functional.

The simplest form of connection establishment is implicit addressing, where a process can only communicate with one other process in the system. This is usually the case for processes created to perform a single service, and only communicate with their parent process. The communication path can then be established a process creation time. Hence, no process or communication path needs to be specified in the other communication primitives for passing messages. Such a model is appealing in its simplicity and can be useful when such restrictive communication is desired. However, it is not powerful enough to allow any pair of processes to communicate, a minimal requirement of a general communication facility.

Explicit addressing is characterized by the explicit naming of the process with which communication is desired. This is proposed in many systems, including Hoare's CSP model [HOAR78] and SUPPOSE [BRIT80]. A specific primitive may be provided for connection

establishment or the process ID can be included in the message passing primitives. The only advantage in providing a separate primitive for connection establishment is in allowing an option to restrict future communications.

Explicit addressing requires a global knowledge source containing the identity of each process in the system. Some systems have predefined names for processes providing system services and a user accessible table of user process ID's. A communication mechanism dependent on explicit addressing is an adequate basis for a complete communication system, as evidenced by its use in the Thoth system [CHER79]. However, explicit addressing is not flexible enough to effectively handle such common circumstances as process migration in distributed systems and multiple processes providing a single service.

The UNIX pipes [RITC74] are an example of an interprocess communication facility using a limited form of explicit addressing. Only processes with a common ancestor may communicate via pipes. Thus, interprocess communication in the original UNIX system is very limited. Extensions of the original UNIX facilities are proposed in [RASH80] and [SUNS77], each using functional addressing (see below).

Path-based addressing associates global names with local message receptacles, or 'mailboxes'. A process can declare a local mailbox into which messages can be received. A process specifies a destination mailbox when sending a message. Since a process can send a message at any time if it knows another's mailbox and the mailbox is in existence, communication paths do not need to be preestablished, and hence are transient. This kind of addressing was introduced by

Balzer [BALZ71] and expanded in work by Walden [WALD72]. The RIG system [BALL76] combines explicit and path-based addressing by requiring the specification of the destination port ID and process ID.

Functional addressing establishes a connection based on the need to serve or request a service. In this case the communication path is itself a named entity of the system. The identity of the process or processes on the other end of the communication path need not be in either the requestor's or the server's view. This kind of addressing is very flexible because an individual process is not necessarily associated with a communication path, and paths themselves may be passed within messages. Functional addressing is the underlying concept used in many current message-based systems, notably Accent [RASH81] with 'port' denoting a path, and DEMOS [BASK77], with 'link' denoting a path. Communication paths established using functional addressing may have a number of characteristics that change over time, including the set of processes with access to them. We will restrict use of the term port to communication paths established using functional addressing.

2.2 Port Types

Typing involves the restrictions placed on the use of a path declared when the path is created. The primary restrictions of concern in connection establishment are directionality, ownership, frequency of use, and transference rights. Additionally, a section to follow is devoted to investigating the restrictions placed on the type of messages that may be passed on a port.

Directionality concerns whether or not a single process has both send and receive access to a single communication path. If so, the path is bidirectional (or duplex, as used in TRIX [WARD79]). Most systems provide only unidirectional (or simplex) paths, on which processes may send or receive messages, but not both, for the lifetime of the path. Bidirectional communication requires a pair of paths in systems providing only unidirectional paths.

Ownership deals with the capability to destroy a port and terminate communication without the consent of all processes with access to the path. Ownership can be associated with the directionality of communication, as in the Accent system where the allocator of a path automatically has ownership and receive access to the path.

Frequency of use deals with the limitations on the frequency a path can be used. The DEMOS and Roscoe [SOLO79] systems have a 'reply' path, created for the sole purpose of returning a single message. Such a path is automatically destroyed after it is used once.

Transference rights concern the ability to duplicate a port, or pass access and/or ownership rights of an established path to another process. For example, a process with ownership rights to a path can send that privilege to a receiving process, allowing it to destroy the path or change the path's characteristics. Transferring access rights can, but does not necessarily, imply loss of the transferred right. Transference rights are relevant to the protection of objects and can also affect the topology of the communication path. Both of these

issues are addressed below.

2.3 Topology

The topology of a communication path is the interrelationship of communicating processes using the path. Topology can take four general forms: one-to-one, one-to-many, many-to-one, and many-to-many. Communication paths that are not one-to-one can often be functionally equivalent to a set of one-to-one paths, an important issue in examining topologically complex path structures. The Process Control Language (PCL) [LESS79] provides a complete specification of a wide variety of topological structures of unidirectional communication paths. The terms used below are introduced in the PCL description.

The simple path is provided for a one-to-one connection. A single queue is maintained to hold all messages. The broadcast and multiple read connections are one-to-many connections, associating a set of receivers with each sender. Every message sent on a broadcast communication path is received by every receiver; any message sent on a multiple read path is received by the first process requesting it. The broadcast path can be easily simulated by a set of simple paths from the sending process to the set of receivers. The multiple read path is useful when a set of servers provide a service, and the service is performed equally well by any of the servers. A multiple-read path cannot be simulated with one-to-one paths without additional management control within the processes performing the service.

Many-to-one connections are provided in the multiple-write and concentration communication paths. A multiple-write mapping is the converse of the multiple-read mapping: every message sent by any sender is received by the receiver. This also can be simulated by forming a set of simple paths from the sender processes to the receiver. The concentration path can be used for synchronization of the set of senders: every message received is the concatenation of the set of messages from a single send by each sender. A message cannot be received until every sender has transmitted a message.

Many-to-many paths are combinations of one-to-many and many-to-one paths. The multiple-write/broadcast transmits every sent message to each receiver. A message transmitted on a multiple-write/multiple-read path by any sender is only received by the first receiver requesting it. A concentration/broadcast message is the concatenated messages from each sender and is sent to every receiver. A concentration/multiple-read message is the concatenated messages from each sender and is received by the first receiver requesting it.

It is not necessary to declare the topology of the communication path at path creation time. For example, a simple communication path between a pair of processes can be transformed into a broadcast path after the path is created when a third process attains receive access to the path. Therefore, the topology of a path is closely related to the transference and access rights of processes to the path, and may change over the lifetime of the path.

3. Synchronization and Concurrency

An essential issue in the discussion of communication primitives, particularly in a distributed environment, is the degree to which they provide concurrency between processes. Primitives providing only synchronous coupling with no parallelism implement, in effect, a message-oriented approach to procedure calls. Communication primitives use process blocking to achieve synchronization. Blocking, the voluntary, temporary suspension of an executing process, prevents maximizing the concurrency between processes. This section examines the varying degrees of synchronization and concurrency provided by different send and receive primitives.

The unconditionally blocking reply-send primitive (remote-invocation send of [LISK79]) has the basic semantics of a procedure call. The invoked procedure is a process blocked at the start of the procedure with a receive, awaiting a message. The procedure caller, or message sender, issues a reply-send to initiate the procedure execution. The contents of the message are the procedure input parameters. The sender blocks until a reply message (the return parameters) is returned by the receiving process, indicating the procedure termination. The receiver blocks by reissuing a receive after the reply-send. (Consult [NELS81] for an extended comparison of message-based and remote procedure call approaches to communication.) Thus, the reply-send primitive synchronizes the sender and receiver at the expense of concurrent process execution.

The synchronized send increases the parallelism between communication processes. With the synchronized send, the sender is blocked until the message is received by the destination process. This primitive is more powerful than the reply-send. It can be implemented on either a simplex or duplex path, and can be used to simulate a reply-send by issuing a blocked receive to await a reply. The sender is also notified immediately of communication failure. The reply-send and synchronized send primitives require no message buffering or queueing if only a single process has send access to the port.

The no-wait send or asynchronous send maximizes concurrency between communicating processes [LISK81]. In this model, the sender resumes execution as soon as the message is composed and buffered within the communication facility. The need for buffering introduces problems of flow and congestion control, addressed in the following section. Additionally, the sender is not notified of transmission errors or communication failure. The no-wait send can simulate a synchronized send if the sender blocks on a receive after issuing the send, and the receiver sends a reply upon receiving the message. The no-wait send is therefore the most flexible of the three send primitives, though it introduces extensive implemention problems. It is the obvious candidate for use in more complex topological path structures. Broadcast and multiple read topologies would result in unacceptable delay without the no-wait send, while use of the no-wait send is implied in the multiple write structure. Most recently developed communication facilities providing asynchronous message passing implement the no-wait send to increase process concurrency.

The test and send primitive tests to ensure a that receiver is blocked awaiting the message before transmitting a message. If the receiver is not blocked, the sender resumes execution and is notified that the message was not sent. The test and send primitive provides the sender with a polling capability, an approach to synchrony different from previously discussed primitives. To our knowledge, no current systems have implemented this primitive.

There are two forms of receives: unconditional and conditional. The unconditional receive, or blocked receive, blocks the receiver until a message is queued on the selected port [RA080,GENT81]. The unconditional receive sacrifices process concurrency by blocking the receiver when no messages are queued.

The unconditional receive has two variations. The first is the port set receive in which the first message received on any of the specified set of ports is returned to the receiver [BRIT80]. This is useful for a serving process awaiting messages from multiple sources. A second variation is the blind unconditional receive which blocks on all ports to which the requesting process has receive access [STEM82]. If a new port is associated with the process after the receive is issued, a message sent to the new port will be returned to the receiver along with the new port ID. This allows processes to respond to the dynamic creation of ports without a special joining mechanism.

The conditional receive (selective receive of [RA080]) introduces a polling capability, allowing the receiving process control over the degree of concurrency desired. The conditional receive polls a (set of) port(s) for a queued message. If a message is present it is

returned, otherwise control is returned with a flag indicating no message is available. The unconditional receive can be generalized to check for a more complex condition than the presence or absence of a message. For example, if messages are prioritized or categorized, the primitive could specify a condition the queued message must satisfy. In the case of many-one communication paths, the condition could specify the source of the message. The Accent system [RASH81] attaches a header to each message and provides a Preview primitive to examine message headers without dequeueing the messages. Combined with the unconditional receive primitive, the Preview primitive can be used to select a specific message from a set of queued messages at a port.

4. Message Delivery

Here we are concerned with issues affecting the delivery of messages. The primary issues are buffer allocation, priority, categorization, and age.

Implementing a general port-based communication facility requires the ability to queue messages at ports awaiting receiver requests. The easiest queueing strategy to implement is first in, first out (FIFO). Regardless of the queueing technique used, queue maintenance introduces the problem of allocating buffers for queued messages. There are three buffer allocation techniques: port-local allocation, process-local allocation, and system-global allocation.

Port-local allocation assigns a fixed buffer space to each port [RASH81]. When the buffer space is filled and a process attempts to send another message to the port, there are three alternatives. First, if a flow control option is included in the send primitive, the port can dynamically expand its size to allow for an additional message. Obviously, there are restrictions placed an a port's expansion size. Second, the sending process can remain blocked until space is available on the port for an additional message. Last, an error flag can be returned to the sending process indicating a full port. The choice of action when a full port is encountered can be an option included in the send primitive, a dynamic attribute of the port, or a static system feature.

The process-local allocation technique associates the buffer space with the process rather than the port [KNOT75]. This restricts the total number of outstanding messages for a process. However, if the processes with access to a port can change throughout the port's existence, it is undesirable to associate message buffers with processes.

A third resource allocation alternative is to maintain a global buffer pool (examples include Roscoe [SOLO79] and UNIX). This technique, termed system-global allocation, introduces the problem of congestion control: assuring that each process has fair access to the communication facility. A viable solution to the problems of flow and congestion control is to combine the port-local and system-global techniques. The port-global hybrid could associate a fixed buffer space with each port while reserving buffer space within the system. Congestion control would then be assured by allotting space to each

port and flow control would be provided at the discretion of the communication facility through the distribution of surplus system space.

Messages may have attributes associated with them to allow for their reception in an order other than the order in which they were sent. Priorities assigned to messages by senders can facilitate the quick delivery of alarm or emergency messages to a receiver. Categories assigned to messages can be used by a receiver to select messages independently of both sender priority and send order. Aging associates a time limit with a message. If the message is not delivered within the specified time limit, the message is deleted from the port queue.

5. Message Structure

There is a great deal of variety in the message structures of proposed and implemented systems. One issue is the message length. The Roscoe, StarOS [JONE79], and Thoth [CHER79] systems have short, fixed length messages. This is in part due to the belief that most messages in such tightly coupled systems are short control messages. Special, synchronous communication paths are provided for large data transfer, such as reading and writing files. Most protocols for loosely coupled systems support variable length message transfers(e.g. Accent, Medusa [OUST80], and CLU [LISK79]). Variable length messages are obviously more difficult to implement due to the buffering problems they induce.

A more important aspect of message structure is the typing of data within messages. Some systems support strongly typed data within messages (Eden [LAZ081], CLU, and Accent). Strong typing of data provides reliability in general, but in a distributed system consisting of heterogenous nodes, it is especially important in assuring proper conversion of messages across nodes.

6. Protection With Ports

We now turn our attention from port-based communication to port-based protection. Towards this end, we first examine capabilities as a mechanism for protecting shared objects and then discuss ports as a technique for implementing capabilities.

6.1 Capability-based Protection Mechanisms

Introduced by Dennis and Van Horn [DENN66], the concept of capabilities was first incorporated into an operating system in the CAL [LAMP76] and Hydra [WULF74] systems, and implemented in commercially available hardware in the Plessey System 250 [COSS72]. Systems employing a capability protection mechanism view all data as objects which are strongly typed and distinguished from one another by unique identifications. Object types are abstract data types because their type defines and limits the operations that can manipulate them. Examples of object operations are create, destroy, duplicate, push, pop, etc. A capability consists of an object identifier and a set of access rights, which allow the manipulation of the object with a subset of the operations defined by the object's type. A process may request an operation on an object only if it possesses the appropriate capability for the requested object.

Capability-based protection mechanisms are difficult to implement. Once a process obtains a capability it is essential that it not be able to modify it. Capabilities may be stored as C-lists, as with the Intel 432 [COX81], or as tagged memory, as with the IBM System/38 [BERS80]. The C-list scheme stores all capabilities in separate capability segments. A major difficulty in the use of capabilities is the separation of data and capabilities. Tagged memory requires each unit of data, 32 bits in the case of the System/38, to be tagged to indicate whether it is a capability or not. Tagging presents a large overhead for memory. See [LEVY81] for a complete description of capablity based architectures.

Object-oriented systems such as CAL, Extended CLU [LISK79], CAP [NEED78], Hydra, and the Intel 432 associate groups of objects with modules. Modules appear in the literature with a variety of names, including domain, context, guardian, and environment. A module consists of a set of processes and local data (both objects themselves) in addition to shared objects. A process within a module has the right to distribute access rights, in the form of capabilities, to processes outside the module. Only processes with a capability for an object have the right to manipulate the object with operations defined in the object's module. The protection of an object is thereby controlled by the distribution of capabilities by processes within a module.

The request for an operation on an object managed by another process is an example of interprocess communication, and thus a prime candidate for implementation with a port-based system. The communication link between processes in separate modules is the port.

Messages passed through ports include object capabilities and operation requests on the objects. The Eden system combines the capability-based architecture and port-oriented communication facility of the Intel 432 to provide an object-oriented approach to protection. Likewise, the proposed Extended CLU system combines the use of ports with a modular design for an object-oriented distributed computing system.

6.2 Proposed Port-based Protection Mechanism

We propose an alternative approach to protection than a strictly capability-based system. In the use of capabilities and ports discussed above, ports provide the means to manipulate objects, while capabilities independently provide protection. In our approach, the capability to manipulate an object using any of a set of operations is equated with the capability to send a request through a port connected to that object's manager. Thus, possession of a port capability implies the right to perform a particular set of operations on the object associated with the port. The communication facility is thereby extended to provide the protection mechanism for all objects in the system.

User processes are permitted to create and thereby own ports to perform a specific set of operations on an object. This permission is granted by an underlying capability manager. To perform a specific operation on an object, a process sends a message to that object's manager through the port created for that purpose. On completion of the requested operation, the result of the operation is sent to the user process. A user process is permitted to request a certain

operation on an object via a port only if it has the capability for the operation. This is achieved by binding the port of each user process to a specific set of operations on a specific object. Thus ports are used to implement a certain capability structure which in turn reflects the desired protection in the system.

Given that ports are used for remote execution of operations, addressing in a system protected via ports is functional. However, one can choose from a variety of send and receive primitives, path topology, message delivery schemes, message-buffering techniques, and path types. Message delivery and message buffering relate only to issues such as performance and hence are outside the domain of protection. Now we briefly discuss some of the decisions to be made in the selection of path types, path topologies, and message passing primitives.

Obviously, given that a port is being used for sending a request for an operation and receiving the results of the operation, a port needs to be bidirectional. Since a port defines the capability of the user that has access to it, definition of ownership of a port and transference rights have important ramifications for protection. One possible choice is to disallow the transference of ownership and capabilities, thus requiring the allocation of new ports when necessary. This restricts paths to a one-to-one topology. The implications of decisions such as this are currently under investigation.

The choice of message passing primitives depends on the intended use of ports. For example, if ports are to be used for remote operation execution only, and no concurrency is desired, then the reply-send primitive seems appropriate. However this in turn can be simulated using other lower level message passing primitives. The implications of such decisions are also being studied.

We should note that a program need not issue the port operations explicitly. It could instead request operations on objects and these requests could then be translated by the system into the appropriate send and receive primitives on ports allocated for the object operations.

Port-based protection appears to achieve the dual requirements of communication and protection uniformly. Currently, a prototype system is being built in order to study the ramifications of the choices to be made in the design and implementation of such a system [STEM82].

7. Summary

We have presented the important aspects of port-based communication. Two salient features of ports deserve highlighting. First, ports use functional addressing, where the identity of the participating processes is secondary to the function served by the communication link. Second, a port is an entity recognized by the operating system independent of the processes it links.

We have introduced a port-based protection mechanism for an object-oriented system. Objects are abstract data types. The ability to manipulate an object requires possession of a capability specifying the object identity and subset of operations that may be used on the

object. We have proposed associating each port with a set of operations on a specific object, thus equating port and capability. Processes access an object by establishing a communication link with the manager of that object. Included in the specification of the communication connection is the set of operations performable on the object. These operations may then be performed by sending messages to the object's manager via the port. We have thus unified the concepts of port-based communication and protection in proposing the use of ports as capabilities.

References

[BALL76] Ball, J. E., Feldman, J., Low, J., Rashid, R., Rovner, P., "RIG, Rochester's Intelligent Gateway: System Overview", IEEE Transactions on Software Engineering, Vol. SE-2, no. 4, December, 1976.

[BALZ71] Balzer, R. M., "Ports -- A Method for Dynamic Interprogram Communication and Job Control", Report for an ARPA contract at RAND, August, 1971.

[BASK77] Baskett, F., Howard, J., Montague, J., "Task Communication in DEMOS", Proceedings of the 6th ACM Symposium on Operating System Principles, pp. 23-31, November, 1977.

[BERS80] Berstis, V., "Security and Protection of Data in the IBM System/38," Proceedings of the 7th Annual Symposium on Computer Architecture, May, 1980.

[BRIT80] Britton, D. E., Stickel, M. E., "An Interprocess Communication Facility for Distributed Applications", Proceedings of the 1980 COMPCON Conference on Distributed Computing, February, 1980.

[COSS74] Cosserat, D. C., "A Capability Oriented Multi-Processor System for Real-Time Applications." Proceedings of the International Conference on Computer Communications. October, 1972.

[CHER79] Cheriton, D. R., Malcolm, M. A., Melen, L. S., Sager, G. R., "Thoth, a Portable Real-Time Operating System", Communications of the ACM, Vol. 22, no. 2, February, 1979.

[COX81] Cox, G., Corwin, W., Lai, K., Pollack, F., "A Unified Model and Implementation for Interprocess Communication in a Multiprocessor Environment", Intel Corporation, 1981.

[DENN66] Dennis, J. and Van Horn, E., "Programming Semantics for Multiprogrammed Computations", Communications of the ACM, Vol. 9, no. 3, March, 1966.

[GENT82] Gentleman, W. M., "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept," Software - Practice and Experience, Vol. 11, pp. 435-466, 1981.

[HOAR78] Hoare, C.A.R., "CSP: Communicating Sequential Processes", Communications of the ACM, Vol. 21, no. 8, August, 1978.

[JONE79] Jones, A. K., Chansler, R. J., Durham, I., Schwans, K., Vegdahl, S. R., "StarOS, a Multiprocessor Operating System for the Support of Task Forces", Proceedings of the 7th Symposium on Operating System Principles December, 1979.

[KNOT74] Knott, "A Proposal for Certain Process Management and Intercommunication Primitives", Operating Systems Review, October and and January, 1974-75.

[LAMP76] Lampson, B. W., Sturgis, H. E., "Reflections on an Operating System Design", Communications of the ACM, Vol. 19, no. 5, May, 1976.

[LAZO81] Lazowska, E., Levy, H., Almes, G., Fischer, M., Fowler, R., Vestal, S., "The Architecture of the Eden System", 8th Annual Symposium on OS Principles, December, 1981.

[LESS79] Lesser, V., Serrain, D., Bonar, J., "PCL: A Process-oriented Job Control Language", Proceedings of the 1st International Conference on Distributed Computing Systems, October, 1979.

[LEVY81] Levy, H., "A Comparative Study of Capability-Based Computer Architectures", University of Washington Master's Thesis, October, 1981.

[LISK80] Liskov, Barbara, "Primitives for Distributed Computing", Proceedings of the 7th Symposium of Operating System Principles, December, 1979.

[NEED77] Needham, R. M., Walker, R. D. H., "The Cambridge CAP Computer and its Protection System", Proceedings of the 6th ACM Symposium on Operating System Principles, November, 1977.

[NELS81] Nelson, B. J., "Remote Procedure Call", Xerox Corperation Technical Report CSL-81-9, May, 1981.

[OUST80] Ousterhout, J., Scelza, D., Sindhu, P., "Medusa: An Experiment in Distributed Operating System Structure", Communications of the ACM, Vol. 23, no. 2, February, 1980.

[POWE77] Powell, M., "The DEMOS File System", Proceedings of the 6th ACM Symposium on Operating System Principles, pp. 33-42, November, 1977.

[RAO80] Rao, Ram, "Design and Evaluation of Distributed Communication Primitives", ACM Pacific 1980, November, 1980.

[RASH80] Rashid, R., "An Inter-Process Communication Facility for UNIX", Carnegie-Mellon University Technical Report, June, 1980.

[RASH81] Rashid, R., Robertson, G., "Accent: A Communication Oriented Network Operating System Kernel", Carnegie-Mellon University Department of Computer Science Technical Report, April, 1981.

[RITC74] Ritchie, D. and Thompson, K., "The UNIX Time-Sharing System", Communications of the ACM, Vol. 17, no. 7, July, 1974.

[SOLO79] Solomon, M. H., Finkel, R. A., "The Roscoe Distributed Operating System", Proceedings of the 7th Symposium on Operating System Principles, March, 1979.

[STEM82] Stemple, D., Ramamritham, K., Vinter, S., "Preliminary Design of a Port-oriented Operating System", COINS Technical report, Dept. of Computer and Information Sciences, University of Mass., Oct., 1982.

[SUNS77] Sunshine, C., "Interprocess Communication Extensions for the UNIX Operating System: 1. Design Considerations", Rand Corperation Publication R-2064/1-AF, June, 1977.

[WALD72] Walden, David C., "A System for Interprocess Communication in a Resource Sharing Computer Network", Communications of the ACM, Vol. 15, no. 4, April, 1972.

[WARD79] Ward, S., "TRIX: A Network Operating System," MIT Technical Report, December, 1979.

[WULF74] Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., Pollack, F., "HYDRA: The Kernel of a Multiprocessor Operating System", Communications of the ACM, Vol. 17, no. 6, June 1974.