INTERPROCESS COMMUNICATION

WITHOUT PROCESS IDENTIFIERS

COINS Technical Report #84-36

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

David Stemple
Stephen Vinter
Krithivasan Ramamritham

INTERPROCESS COMMUNICATION WITHOUT PROCESS IDENTIFIERS

[1]

David Stemple
Stephen T. Vinter
Krithivasan Ramamritham

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

ABSTRACT

An interprocess communication facility provided by the kernel of an experimental operating system is presented. The operating system, called the Gutenberg system, has been designed in order to study the interrelated issues of object-based protection and support for abstract database types. In the Gutenberg system, processes communicate via communication channels called ports. A process may create a port and have it connected to a second process if the first process has the privilege to request one of the second process' services, such as an operation on an object managed by the second process. The port is typed by the service it can be used to request and is created by reference to its function without using the identifier of the process providing the service. By using this manner of port creation with its functional process addressing, interprocess transfer of port privileges, and the new concept of cooperation class introduced in this paper, arbitrary process interconnection topologies can be achieved without any explicit use of process identifiers by processes.

In this paper, we review process addressing in interprocess communication facilities and then present the Gutenberg approach to the creation and interconnection of processes. Examples of object sharing with abstract data type managers and data-driven protocols of database query execution are used to illustrate the methods of constructing systems of cooperating processes using the Gutenberg system. Issues involving functionality, complexity, and achievable process interconnection topologies using the various Gutenberg mechanisms are explored.

---

[1] This material is based upon work supported in part by the National Science Foundation under grant MCS 82-02586.

1. INTRODUCTION

As more attention is paid to distributed modes of computing, new techniques for interprocess communication continue to be developed. These developments are taking place in programming languages [DOD83, FELD79, BRIN78, LISK79, COOK80], operating systems [RASH80, RASH81, GELE82, LANT80, KAHN81, SUNS77, VOYD80], hardware architecture [COX81], and communication models [GENT81, NELS81, SPEC82]. In operating systems much work has been done on message-passing protocols used for interprocess communication. Important issues in message-passing protocols include communication channel topology, channel establishment, process addressing, synchronization, and message buffering. In this paper we will concentrate on one method of channel establishment and process addressing, and explore its ramifications in functionality, complexity, and achievable topologies.

The Gutenberg system is an experimental operating system designed to allow all interprocess communication to be organized as requests to abstract data type managers. These requests are communicated via channels called ports. Though the interprocess view is object-oriented, the intraprocess view used by programming languages running on the system need not be: one goal of the design is to make high-level, object-oriented views of remote services (e. g., operating system services and database operations) available to all programming languages on the system. This requires that the kernel be involved in process interconnection but not in local procedure calls. Thus we separate the control of process interconnection from programming language features which are provided mainly for the static, compile-time

control of procedure (or module) interconnections [TICH79]. This non-uniformity in the system's object orientation derives from a number of considerations, some dealing with human factors considerations, others with efficiency.

While we take a different approach to module interconnection control from that of DeRemer and Kron [DERE76], we agree with their contention that "structuring a large collection of modules to form a system is an essentially distinct and different intellectual activity from that of constructing the individual modules." Therefore, we believe that the separation of process interconnection control from programming language facilities is justified because it is "natural" to use a distinct and different tool for a distinct and different activity, and because it provides for secure but dynamic control of interconnections more simply than programming language methods can.

A major purpose of module interconnection control is to control the access to shared objects. *Capabilities* [DENN66] are one means of dynamically controlling access to shared objects. The work reported in [KIEB78, MCGR79, and ANCI83] incorporates capabilities into programming languages in an attempt to increase the flexibility of controlling dynamic access to shared objects. Capability-based operating systems, e. g., Plessey 250 [ENGL74], Hydra [COHE75], and iMAX [KAHN81], provide for dynamic control through the passing of capabilities for accessing modules. Capabilities are means for providing the control implicit in the abstract data type paradigm, i.e., the limiting of object access to a set of operations defined by the object's type, and have demonstrated the ability to facilitate the production of reliable systems, e. g., using the Plessey 250. To date, however, capabilities have been expensive to implement either in software or hardware, and need further development

before they can demonstrate their cost-effectiveness when exploited in a general and uniform manner. The main problem seems to be their use both for accesses to independent modules and for accesses to local data and procedures. The Gutenberg system represents an attempt to avoid this problem by using a capability scheme for controlling access to shared objects (interprocess communication control), but not for access to local objects, e. g., via local procedure calls.

In addition to choosing a non-uniform use of capabilities, we decided early in the project that a process would never use another process' identifier to establish a communication link with it, but instead would use the name of an object type whose instances were to be delivered on a port bound to the type. This decision was motivated by the belief that the identification of processes is a low-level function not appropriate in the building of well-structured systems of cooperating computations, and that the distribution of processing and data would be simpler in a system which avoided explicit identification of processes at the user process level. In this paper we present those features of the Gutenberg design which result from the avoidance of explicit process identification in port creation, the use of ports for all shareable object access, and the non-uniform use of capabilities.

The avoidance of process identifiers led to the development of a novel manifestation of the concept of cooperation, namely the *cooperation class*. A cooperation class is a capability assignable by user processes to different kinds of cooperation, such as two processes sharing an object or two processes providing input to a common third process. We feel that cooperation class is a single mechanism corresponding to a single concept, *cooperation*, that has traditionally been implemented by multiple mechanisms, such as process identifiers, object identifiers, and

semaphores, none of which is used in Gutenberg. This paper provides two examples of the use of cooperation class to accomplish two different kinds of cooperation.

The primitive elements of the Gutenberg system are ports, conceived as instances of an abstract data type whose manager is the operating system kernel; the *capability directory*, a directory of port creation capabilities specifying the types of objects which can be sent and received on ports; processes; and *cooperation classes*, used to partition processes into cooperating, communicating groups. The kernel manages processes in response to port operation requests and is guided by the capability directory.

While ports are not used by a process to communicate with the kernel, they constitute the sole means by which one process can affect another. However, a port is created for a specific function, not for communicating with a specific process. In fact, there is no way for a process to address another process directly. The purposes for which ports may be created are contained in the capability directory as types of objects to be sent or received on the ports. Type descriptions in this directory are connected to the definitions (initial images) of the processes which will perform the functions for which ports are created. The capability directory is process independent and updatable in the same way that a file directory is, and, at any time, represents all currently allowable interprocess communication. This is a novel approach to the storage of capabilities and has ramifications in the area of process cooperation, the subject of this paper, and in the related problems of protection, touched on only lightly in this paper.

We will show how the Gutenberg paradigm of system organization can be used to realize well-structured implementations of both remote procedure calls and interprocess communication networks of arbitrary topology. We will also introduce the concept of cooperation class and show how it can be used to implement the sharing of objects between many processes and data-driven protocols without requiring process identifiers. Relational database query execution provides a concrete example of a multi-process computation for purposes of illustrating a data-driven protocol.

The remainder of this paper is organized as follows. The next section is a review of interprocess communication methods with emphasis on process addressing. Following this is an introduction to Gutenberg interprocess communication techniques in a master/slave model which includes remote procedure call semantics as a special case. Next we introduce port passing and other Gutenberg methods of breaking down the master/slave hierarchy and allowing arbitrary interconnection topologies. Then, we show that a relatively simple concept of cooperation class can significantly reduce the complexity of implementing protocols encountered in the context of shared remote objects and distributed database queries. Finally we present a summary and a discussion of further work to be done.

2. PROCESS ADDRESSING IN INTERPROCESS COMMUNICATION FACILITIES

There are different methods that can be used to address processes with which a process desires to communicate. We have chosen to categorize process addressing as implicit, explicit, path-based, or functional [VINT83]. 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 which only communicate with their parent process. Though such a model is appealing in its simplicity, it is not flexible 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]. This addressing scheme is particularly suited for process configurations in which the output of one process serves as input to another. 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 identifiers. A communication mechanism dependent on explicit addressing is an adequate basis for a complete communication system, as demonstrated by its use in the Thoth system [CHER79] and MSG [NSW76] used in the National Software Works. MSG extends explicit addressing by supporting two kinds of explicit addressing: *generic* and *specific* addressing. Generic addressing allows a process to specify an explicit name which is used to connect to an available process from a collection of indistinguishable processes, while specific addressing identifies a single process. However, explicit addressing is not flexible enough to effectively handle such common circumstances as process and path migration in distributed systems.

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).

Global addressing associates global names with local message receptacles, or "mailboxes". A process can declare a local mailbox into which messages can be received and specify a destination mailbox when sending a message. This kind of addressing was introduced by Balzer [BALZ71] and expanded in work by Walden [WALD72]. Such a scheme is used in [GOOD79] wherein the mailbox is global to the processes that use it. The RIG system [BALL76] combines explicit and global addressing by requiring the specification of the destination port ID and process ID. A form of global addressing using a distributed but globally accessible memory for process communication is described in [GELE82].

Functional addressing establishes a connection based on the need to serve or request a service. In this case the communication path and possibly its type are named entities of the system. For the user of a path, the identity of the process or processes on the other end of the path is insignificant. What is significant is that the correspondents are providing or requesting a service. This kind of addressing is 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 the following: PLITS [FELD79] where "slot" denotes a path, STARMOD [COOK80] and Communication Port [MAO80] where "port" denotes a path, Distributed Processes [BRIN78] and Ada [DOD83] where "entry" denotes a path, and SR [ANDR81] where "operation" stands for path. It is also used in the following message-based operating systems: Accent [RASH81] with "port" denoting a path, and DEMOS [BASK77], with "link" used for a path.

In the following sections we examine the use of one form of functional addressing in interprocess communication, and investigate its properties from the point of view of supporting an abstract data type approach to all of a process' contact with its external environment.

3. MASTER/SLAVE, USER/SERVER CONNECTIONS

The basic process interconnection method of the Gutenberg system uses a form of functional addressing in which the function is the sending or receiving of objects of a named type. Process interconnection is accomplished by a port created for the explicit purpose of either sending or receiving objects of one named type, referred to as the port's object type. The port creator supplies the name of the object type and the send/receive choice, but not the name of the process to be connected to.

In the simplest interconnection protocol, creating and using a port causes a new process to be created from an initial process image which is connected to the port's object type name in the capability directory. The new process becomes the *server* of the port.

As an example, consider a process that needs to receive a weather report. In order to accomplish this, the process must create a port for that purpose and then execute a RECEIVE on the port. A subdirectory in the capability directory containing the name of the weather report object type, say WEATHER, would have to have been created previously, and, at the time of the port's creation, would have to be the process' *active subdirectory* (NEWSDIR in figure 1). The active subdirectory of a process contains the names of object types currently available to the process for the purpose of creating ports. Figure 1 illustrates the situation

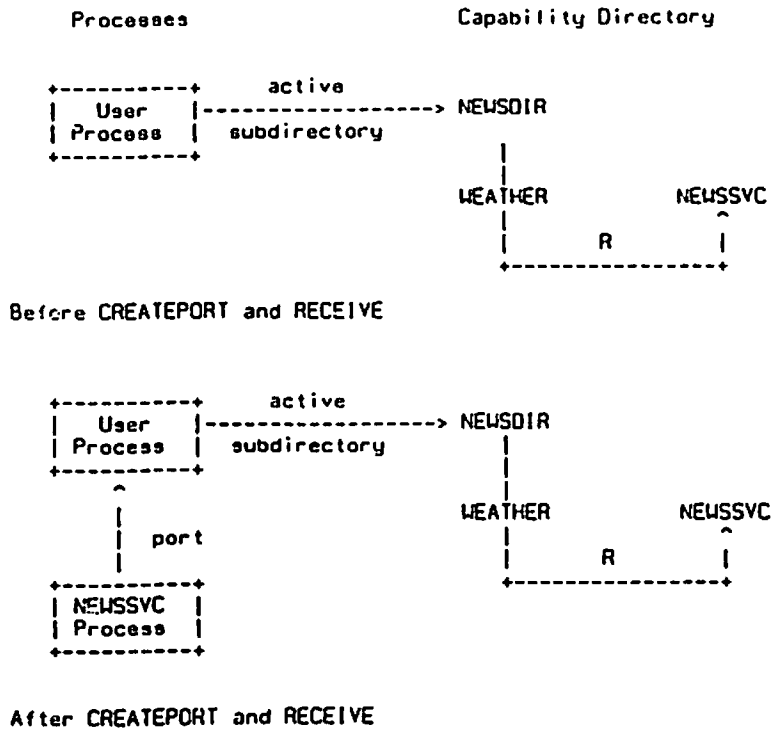

Figure 1: Creation of the NEWSSVC process by CREATEPORT and RECEIVE.

---

before and after the CREATEPORT and RECEIVE operations. The NEWSSVC node, which identifies the initial image to be used in creating the port server process, is not necessarily visible to the user process. In any case, its name is not supplied in the CREATEPORT or RECEIVE operation requests.

The NEWSSVC process would normally get the weather report from a file, though other methods, including access to a remote processor in a network or even random generation (a technique used often in New England), are possible. In fact, the NEWSSVC node could be replaced by a reference to the file manager and a RECEIVE on the port would then be implemented by a simple record read of a file named WEATHER. Thus, ports can be used to redirect I/O requests in the manner of UNIX pipes [RITC74] and to hide details of a service implementation.

The RECEIVE in the example above, if blocking is chosen as the synchronization option when it is executed, has the basic semantics of a remote procedure call [NELS81] of NEWSSVC with the character string WEATHER as an input argument and the weather report as an output parameter. In order to handle cases where the parameters are more complex, Gutenberg includes the SELECTRECEIVE operation with which data may be both sent and received in one execution. The data sent is called *request details* and is used by the port server in serving the request. Among the request details may be port use privileges and other capabilities which can be used as essentially call-by-reference and function parameters by the server process, though no shared memory is implied. Since SELECTRECEIVE may be executed as a blocking operation, the simplest process reference and creation protocol includes remote procedure call semantics in a single port operation. However, port privilege passing and non-blocking synchronization options make the SELECTRECEIVE operation more robust and flexible than remote procedure calls.

4. NON-HIERARCHICAL INTERCONNECTION TOPOLOGIES

Though the master/slave model is useful in many cases, a facility to be used in building general systems of cooperating processes must allow more complex interconnections, including full two-way communications between two autonomous processes. In this section we present one technique for meeting such requirements without giving up the design constraint that prohibits processes from using other processes' identifiers for any purpose. First we look at multi-port servers, and then at arbitrary topologies achievable through multi-port servers using port privilege passing.

There are three process initiation protocols in the Gutenberg system. In the simplest protocol, outlined in the previous section, a new process is initiated as a part of the first operation on a port. Initial process images which cause the kernel to follow this protocol are called *creative*. At the other extreme are the *conservative* process images. A conservative image is used to create a process only if the system contains no process which was created using the image. The kernel makes this determination on executing the first operation on a port. If a process spawned from the image already exists the new port is attached to it. Otherwise, a new process is created as in the creative case. A third intermediate protocol allows new processes to be spawned selectively based on a parameter in the CREATEPORT call. We delay discussion of this protocol until the next section.

Conservative process images provide one means of producing multi-port server processes. A file manager handling the top level interface between user processes and a disk handler is a good example of the need for multi-port server processes. Any process creating a port for the purpose of sending or receiving records of a file whose name (the object type) is linked to a conservative file manager image would have the port connected to the single file manager process. The file manager process could then field all requests and coordinate access to the disk manager as necessary.

Conservative processes expand the attainable process interconnection topologies considerably, since any two processes can communicate indirectly through a conservative process to which they are both connected. However, it is impossible to establish direct, full duplex intercommunication between two arbitrary processes using just creative and conservative processes. If, however, processes are allowed to pass port use privileges via ports, then arbitrary topologies are attainable.

In order to establish two ports between two independent processes, we can use a conservative process which serves both independent processes and passes the server ends of two ports to the two independent processes, making them complementary owners and servers of the two ports. This creates a cycle in the owner-server interprocess relationship which is established by ports. Both processes must have a directory similar to that shown in figure 2 as their active directory while establishing the interconnection.

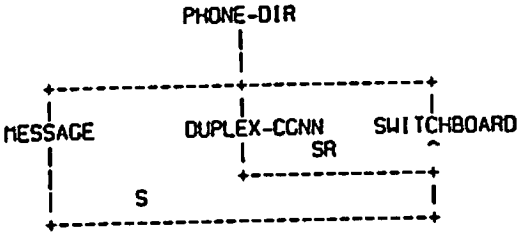

Figure 2: Subdirectory used in establishing two-way communication.

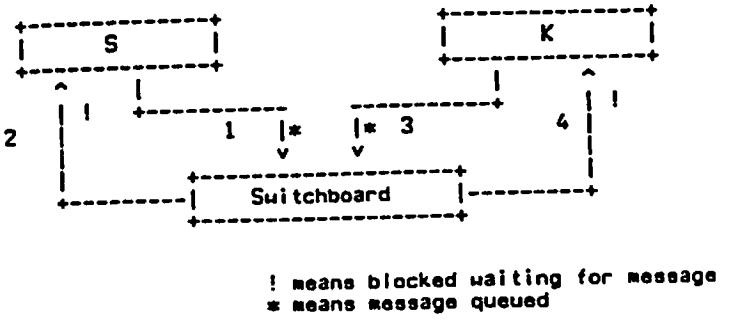

Figure 3: Processes and ports before Switchboard has served requests.

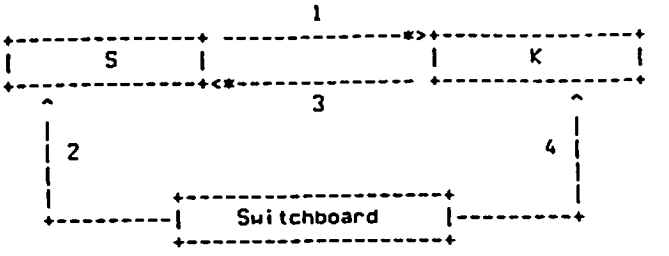

Figure 4: Processes and ports after Switchboard has served requests.

Below are the calls the two processes which identify themselves as S and K could make to establish the interconnection. The names S and K are chosen by the human users and have no relationship to the system process identifiers. See the discussion below for ways in which these names could be validated, none of which involve system process identifiers. (Note: VAR:=*value* in a parameter list indicates that VAR is set to *value* by the procedure called.)

```
S:
 CREATEPORT(PORTID1:=port_1, 'MESSAGE', 'SEND', STATUS)
 CREATEPORT(PORTID2:=port_2, 'DUPLEXCONN', 'SELECTRECEIVE',STATUS)
 SEND(PORTID1, 'Greetings to K!', STATUS)
 SELECTRECEIVE(PORTID2, ACKBUFFER,'From:S, To:K, P:port_1',
    'WAIT', STATUS)

K:
 CREATEPORT(PORTID3:=port_3, 'MESSAGE', 'SEND', STATUS)
 CREATEPORT(PORTID4:=port_4, 'DUPLEXCONN', 'SELECTRECEIVE',STATUS)
 SEND(PORTID3, 'Greetings to S!', STATUS)
 SELECTRECEIVE(PORTID4, ACKBUFFER,'From:K, To:S, P:port_3',
    'WAIT', STATUS)
```

After executing these commands the process/port connections would be as shown in figure 3. The switchboard process would execute the following to establish the

interconnection desired.

```
Switchboard:
 ACCEPTREQUEST(OBJTYPE1:='MESSAGE', PORTID1:=port_1, OP1:='SEND',
    'WAIT', STATUS)
 ACCEPTREQUEST(OBJTYPE2:='DUPLEXCONN', PORTID2:=port_2, OP2:=
    'SELECTRECEIVE', 'WAIT', STATUS)
 GETDETAILS(PORTID2, REQDET1:='From:S, To:K, P:port_1')
 ACCEPTREQUEST(OBJTYPE3:='MESSAGE', PORTID3:=port_3, OP3:='SEND',
    'WAIT', STATUS)
 ACCEPTREQUEST(OBJTYPE4:='DUPLEXCONN', PORTID4:=port_4, OP4:=
    'SELECTRECEIVE', 'WAIT', STATUS)
 GETDETAILS(PORTID4, REQDET2:='From:K, To:S, P:port_3')
 SEND(PORTID2, port_3, STATUS)
 SEND(PORTID4, port_1, STATUS)
```

Figure 4 illustrates the situation after the switchboard process has executed the operations above.

The SEND on port_2 and port_3 would unblock S and K respectively after which the following commands could be used to receive the greeting messages.

```
S:
 RECPORT := Port_part(ACKBUFFER)
 RECEIVE(RECPORT, MESSAGE:='Greetings to S!', 'WAIT', STATUS)

K:
 RECPORT := Port_part(ACKBUFFER)
 RECEIVE(RECPORT, MESSAGE:='Greetings to K!', 'WAIT', STATUS)
```

More messages could be sent by S (and similarly by K) by executing operations of the form:

```
 SEND(PORTID1, 'How are you?', STATUS)
```

The switchboard process could be programmed to enforce a password scheme for K and S to use in their identification, or the system identification of users based on their password-protected logon directory could be used as in many systems. The major appeal of the scheme presented is that no special feature of the kernel needs to be dedicated to it. Many such services with different features could exist in the system without requiring specific support in the operating system kernel.

It should be clear that any arbitrary topology can be attained using port passing and conservative interconnection service processes. However, in certain cases, such a technique is more complex than need be, and a simpler method would be better. In the next section we give examples of problems not easily amenable to such a technique, and present the Gutenberg feature, cooperation class, which allows a simpler solution to these problems without violating the design constraints.

5. SHARING REMOTE OBJECTS AND COOPERATION CLASSES

In this section we introduce the concept of **cooperation class**. We introduce it in the context of sharing remote objects. A remote object is an object managed by a process other than the user process. The effect of using cooperation classes is to control the initiation of new processes and the connection of ports to manager processes. We first discuss the different kinds of managers one could implement using the Gutenberg system, and then show how process initiation protocols and cooperation classes are used to control the access to manager processes and thus control the sharing of remote objects.

In the example in section 3, WEATHER is seen by the user process as an inventory of weather reports from which a report may be received on a port. The manager of this inventory, NEWSSVC, is normally nameless to the user process though its name must appear in some subdirectory. WEATHER and other types connected to manager nodes in this way are types of objects on which the only meaningful operations (from an interprocess point of view) are RECEIVE, SEND, or SELECTRECEIVE on a port. Managers which deliver and accept objects of such types are called *inventory managers*, since they manage an inventory of the objects of the type.

Resources or objects managed externally to a process and never sent or received on ports are, nevertheless, accessible via ports. The access ports for such objects are typed with the resource/object operation names, and the objects which are actually sent and received on these ports are the operation requests, arguments, and acknowledgments. The operation names form the accessing process' view of the remote object. Thus this approach to port typing implements an abstract data type or object-oriented remote access facility.

A manager which services ports typed with operation names is given the operation name when notified of a port's existence. The manager then performs the operation and uses the port to send the requester an acknowledgment, and output in the case of a functional operator such as a stack pop. Managers which manage objects in response to operation requests received on ports are called *abstract datatype managers*, or *ADMs*. Figure 5 gives the subdirectory expressing the view of a stack abstract data type whose manager definition is represented by the node STACK.MGR.

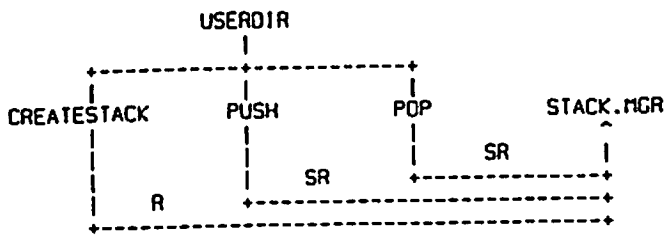

Figure 5: View of the stack abstract data type.

In order to create and use a stack, a process first creates a port for the purpose of requesting a CREATESTACK operation. When the process executes a RECEIVE on the CREATESTACK port, the kernel must determine which process to connect the port to, and whether or not a new process for the purpose needs to be created based on the kind (i. e., creative or conservative) of process image. We will first show that both creative and conservative manager process images are inappropriate for supporting multiple instances of shared abstract data type objects (stacks in the example), and then present the Gutenberg solution to this problem.

Let us assume first that STACK.MGR is a creative process image. When the user process executes the RECEIVE on the CREATESTACK port, a new STACK.MGR process is created and given the request. The new STACK.MGR process initializes a new stack in its local memory and sends an acknowledgment to the requester on the port. Since the manager process is creative, each request to push or pop will cause a new stack manager process to be created. This will not work since the stack resides in the original stack manager process and is not accessible to the new processes.

If the STACK.MGR is a conservative image, then a single stack manager process manages all stacks. The manager manufactures identifiers for the individual stacks and sends them back to the creators via CREATESTACK ports as acknowledgments. These identifiers can be used subsequently to request pushing and popping of the appropriate stacks. While this may be appropriate for some ADMs, it has two major drawbacks. First, it requires all managers of shared objects to be written so as to manage more than one object. Such managers would be more complex than managers using separate process instantiations to isolate their single objects. The second disadvantage of using conservative ADMs is in the protection of shared objects. Without identifying requesters there would be no way for a manager to protect objects from unauthorized use since the object identifiers would be forgeable.

In order to solve the problem of sharing and protecting access to remote ADM objects as simply as possible, Gutenberg makes use of *cooperation classes* and a third process initiation protocol, and further extends the ability to pass information to include cooperation class passing, in addition to port passing.

The third process initiation protocol, intermediate between creative and conservative, allows new processes to be spawned selectively based on a parameter supplied at port creation time. This value, which is used to form groups of cooperating processes, is called a *cooperation class identifier*, or simply a *class-id*. A class-id is a unique value supplied by the system on demand and is unforgeable by virtue of system-maintained lists of processes' class-ids similar to C-lists in certain capability-based systems (e.g. [COHE75]). The class-id is a capability, though it does not identify any particular object or operation. It is a capability to participate in a

particular cooperation.

A process may specify one of its class-ids when creating a port. This results in the kernel associating the class-id with the port's server process upon initiating the process. If the server process image is of the third type, *class-conservative*, then the third process initiation protocol is used. In this protocol, one of two actions occurs when the first operation on the port is performed. If there is no current process spawned from the server image and associated with the port's class-id, one is created and the port is attached to it. If such a process exists, the port is attached to it as in the conservative case.

In the example of the stack manager, if the manager image is class-conservative then PUSH and POP requests on the same stack can be routed to a single STACK.MGR process (designed to manage one stack). The creator of the stack first requests a unique class-id from the system, and then associates it with a CREATESTACK port. When the creator executes a RECEIVE on the port a new stack manager process is created and associated with the port's class-id. Later, in order to push or pop this process' stack, a port is associated with the class-id as well as with the PUSH or POP type node in the capability directory. Executing SELECTRECEIVE on this port connects the port to the pre-existing manager process. This process serves the request by either using the request details of the SELECTRECEIVE as the element to be pushed (PUSH), or by sending the popped element to the requester via the port (POP).

In order to share access to the stack, the creator can pass the class-id to other processes via ports. The other processes can then associate the class-id with PUSH or POP request ports, and the manager of the stack identified by the class-id will get the requests on those ports. The creator can also combine a class-id with a particular object type, say POP, and pass the combined capability to another process, thus permitting the second process to use the class-id only for the purpose of popping the stack identified by the class-id. In order to enforce this discipline, class-ids are created only by the kernel and kept in capability lists maintained by the kernel as part of process states. When a class-id is sent on a port, the sender must have the class-id in its capability list and the capability must include the right to pass it on a port. Upon receiving a class-id, a recipient process' capability list is updated by the kernel to include the class-id.

Class-ids, whether simple or combined with object types, can also be registered in subdirectories to create stable protection domains for processes [STEM84]. By such means, as well as by the passing of port privileges, access to remote objects can be controlled and distributed. A capability-based scheme for passing and revoking privileges for the use of ports, classes, and subdirectories is described in [RAMA83].

In the example above the class-id looks and acts very much like a process identifier, but is, in fact, a more general entity. It can be used for many purposes: for identifying objects (virtual objects as well as real), for choosing communication partners, for constructing networks of cooperating processes, for connecting more than one port to another process, for attaching privileges to groups of users, and even as timestamps for cohort processes in a database transaction (if a certain discipline in generating new class-ids is used) [DATE83]. A class-id may be

associated with any number of processes and may even be registered in a subdirectory as a process-independent capability [STEM84]. We believe that applying cooperation classes to port-based communication represents a new technique which can be used to facilitate the implementation of well-structured systems of cooperating processes. In the next section we give another example of cooperation class usage to demonstrate further its utility.

6. DATA-DRIVEN PROTOCOL EXAMPLE

We conclude with an example which demonstrates how a data-driven protocol for database query execution could be implemented using the Gutenberg facilities. We refer the reader to [STEM83] for a discussion of demand-driven protocols and the distribution of processing across nodes in a network.

This example involves a user process requesting tuples (records) which contain information about clerks who work in Detroit departments. A query has been previously compiled by the query system and named DETCLERKINFO. The query is visible to the user process as an object type name DETCLERKINFO in the process' active subdirectory, and tuples generated by the query are requested as if they were records from a file named DETCLERKINFO.

The relational algebra tree representing this query is given in figure 6. This tree contains two join nodes, each of which will be implemented by a process. Since the join processes will receive input tuples via ports from two separate pairs of processes, it is important to connect the ports correctly. This example shows how cooperation classes can be used to establish the proper communication topology for the execution of such a tree.

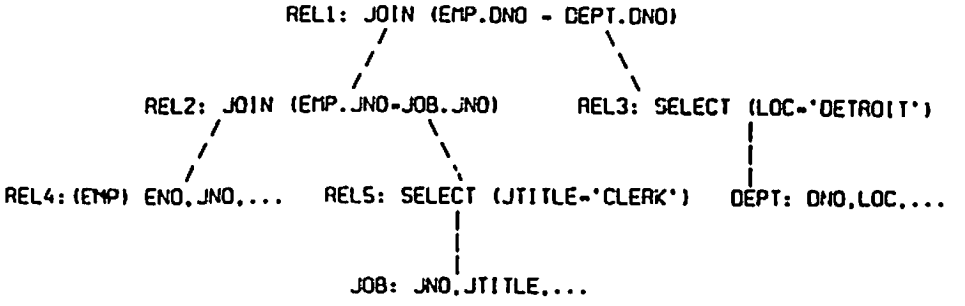

Figure 6: The query tree.

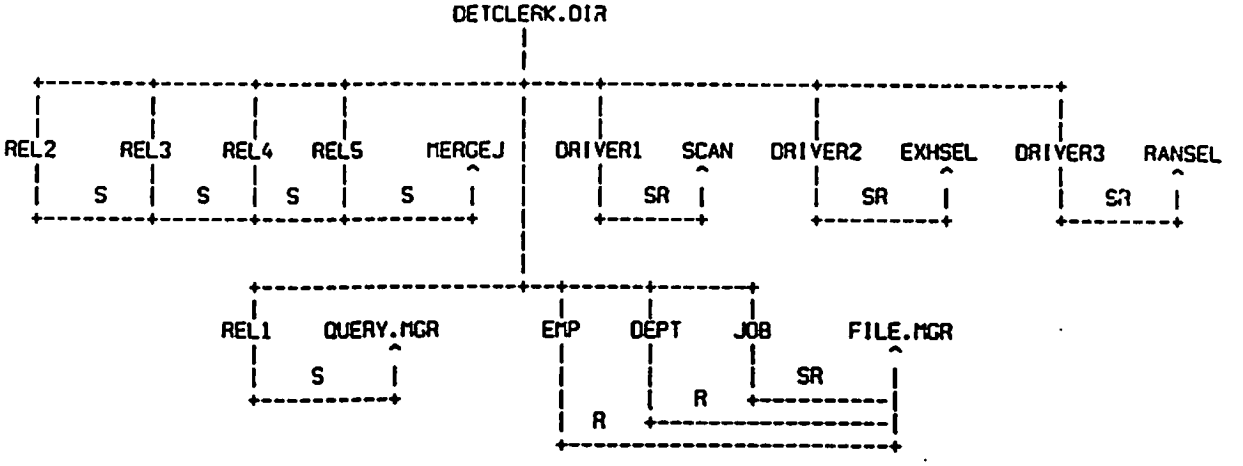

Figure 7: Subdirectory used in executing DETCLERKINFO query.

The query compilation generates two kinds of information used to direct the execution of the query: the *query subdirectory* and the *execution parameters*. The generation of the query subdirectory, DETCLERK.DIR in figure 7, itself has two phases. First, it is necessary to register manager definition nodes for the algorithms to be used in the execution, e. g., RANSEL and MERGEJ. Second, an object type node must be registered for each node in the query tree, e. g., EMP, REL2. These nodes will be used to create the ports connecting the algorithm processes. The names

denote the type of objects to be transmitted over the ports. EMP, DEPT, and JOB will be used to create ports for transmitting records from the file manager to beginning processes (the drivers) of the query execution.

The execution parameters, the second part of the information needed for directing the execution of the query, are passed to the driver processes when they are initiated by the query manager. These parameters are used to supply each process with the information it needs to establish the proper connections to other processes, e. g., an output type name, and to execute its algorithm, e. g., the predicate for a select algorithm.

The user process starts the query execution by creating a port for the purpose of receiving DETCLERKINFO tuples, and executing a RECEIVE on the port just as if it were reading a file named DETCLERKINFO. The active subdirectory of the user process must contain the DETCLERKINFO node connected to the process image of the query manager. The query manager, which we will assume is conservative, looks up DETCLERKINFO in its compiled query table, and gains access to the query subdirectory shown in figure 7 and the execution parameters shown below.

```
DRIVERS = 3;

DRIVER1: (INPUT = EMP; OUTPUT = REL4; CLASS = C1)
PASS: ((OUTPUT = REL2; CLASS = C2;
 PREDICATE = [EMPJNO = JOBJNO])
 PASS: (OUTPUT = REL1; CLASS = C1)

DRIVER2: (INPUT = JOB; OUTPUT = REL5; CLASS = C1;
PREDICATE = [JTITLE = 'CLERK'])
PASS: ((OUTPUT = REL2; CLASS = C2;
 PREDICATE = [EMPJNO = JOBJNO])
 PASS: (OUTPUT = REL1; CLASS = C1)
```

DRIVER3: (INPUT = DEPT; OUTPUT = REL3; CLASS = C2;
PREDICATE = [DEPT.LOC = 'DETROIT'])
PASS: (OUTPUT = REL1; CLASS = C1)

The query manager starts each of the three drivers using a SELECTRECEIVE of the driver name, e. g., DRIVER1, and passes the corresponding execution parameter as request details. The driver processes are initiated from the initial process images for the algorithms in the leaf nodes of the query tree. For example, the process initiated as a result of the SELECTRECEIVE of the DRIVER1 name is named SCAN for the scan algorithm that is responsible for getting employee records from the file manager. Upon initiation, each driver process creates an input port typed with the name from its INPUT execution parameter. This name is the same as an object type name in the DETCLERK.DIR subdirectory and causes the port to be attached to the right process; in all three cases, to the FILE.MGR (file manager) process. For example, DRIVER1 will have its input port typed EMP which is linked to FILE.MGR, the initial process image for the file manager, in the DETCLERK.DIR subdirectory. In Figure 8 we see the configuration of ports and processes after all three DRIVER processes have established ports to the file manager.

Note that the RANSEL (for random access select) process, which uses random access reads of tuples to satisfy its predicate, was chosen during query compilation by the optimizer based on costing heuristics and available access paths. The EXHSEL (for exhaustive scan select) process executes a different select algorithm, also chosen by the optimizer. If such optimization was not used, then the same algorithm could have been used to implement both select operations. In this case cooperation classes could be used to properly generate and connect ports to the processes that execute

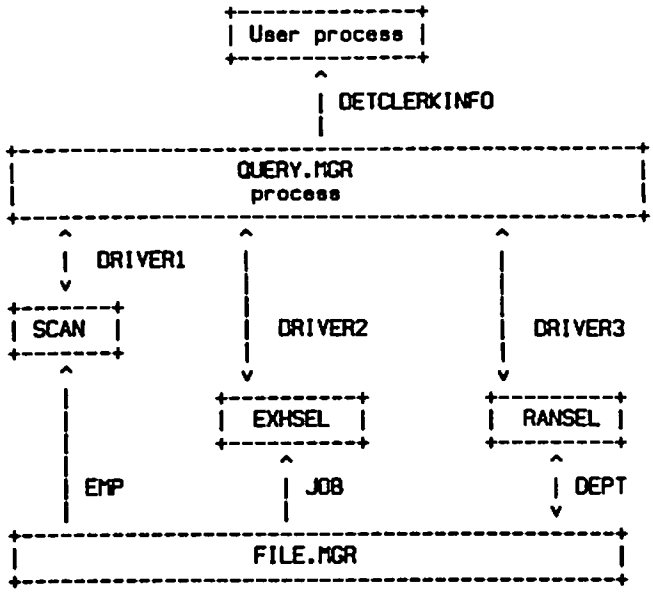

Figure 8: Ports and processes after drivers have started.

the same select algorithm (as is currently the case with processes executing the MERGEJ join algorithm).

The driver processes SCAN, EXHSEL, and RANSEL next use the OUTPUT field of their execution parameters to create ports on which to send their output tuples, typed REL4, REL5, and REL3, respectively. The first SEND operations on these new ports, when the first tuples have been generated, result in the instantiation of the MERGEJ processes using the MERGEJ initial process images. As was the case with the input field, these tuple names correspond to object type names in the DETCLERK.DIR subdirectory. Unlike the ports created to the file manager, these ports have associated cooperation classes, which ensure that exactly two MERGEJ processes are created and that SCAN and EXHSEL are linked by a port to one MERGEJ process while RANSEL is linked to the other. These process connections are evident in figure 9, where ports are labelled by the types of tuples that are transferred over them (e.g., REL4), followed by the associated class-id, if any, in

parentheses (e.g., C1).

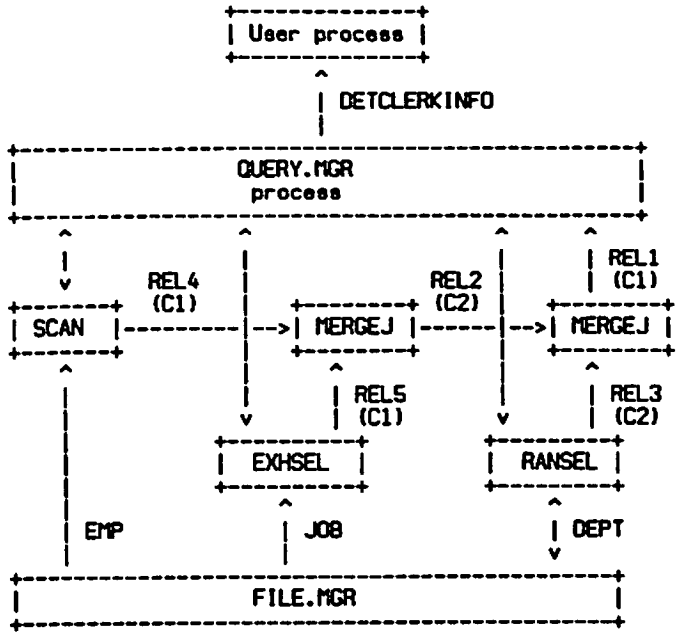

Figure 9: Ports and processes after tuple has been produced.

The two MERGEJ processes are initiated and connected to properly as a result of the use of class-ids and the fact that MERGEJ is a class-conservative process image. Remember that there would be four MERGEJ processes if the MERGEJ process image was creative, and one if it was conservative.

A process in the query execution which receives execution parameters containing a PASS field is required to send the contents of the PASS field through its output port, along with the first tuple the process produces. Since the PASS field may itself contain other INPUT, OUTPUT, CLASS, and PASS fields, execution parameters are propagated up the process execution tree. This results in both MERGEJ processes receiving execution parameters which determine their output ports and the ports' associated classes. The MERGEJ process which receives tuples of type

REL4 and REL5 creates a port of type REL2, with associated class C2, that connects to the second MERGEJ process. The second MERGEJ process creates a port of type REL5, with associated class C1, that is used to send the tuples that satisfy the query back to the query manager. The class-id C1 is not used to choose a query manager process since the query manager is conservative, but is used to tell the query manager which execution of the DETCLERKINFO query, (there could be several), it is receiving tuples from on the port.

In figure 9 we see the final form of the process interconnection. This stage is reached at the time the query manager receives from the REL1 port the first tuple it can send to the user. Subsequent tuples are produced without requests since the driving routines are designed to produce all of their output after being started. The data-driven part of the execution is from the drivers to the query manager. The query tuples being produced will be sent to the user process on its requests as they are issued. This could be changed so that a user process was itself data-driven by associating it with a query during compilation time, and then initiating the query from another process, say the terminal command interpreter process.

Obviously, a long series of sends on a port with no matching receives could cause severe buffering problems. This problem can be solved by the simple expedience of blocking senders whenever a port fills to some level. This could happen to any of the processes in the query, including the driving processes. This problem is a side effect of the decision to maximize asynchronicity of the cooperating processes. However, we feel the provision for asynchronous communication in Gutenberg can make a significant contribution to performance in some applications. In such applications, a remote procedure call mechanism would be

inadequate.

The avoidance of process id's facilitates the asynchronous execution of processes in this data-driven example. This is demonstrated by the SCAN and EXHSEL processes, which need to establish a connection to a MERGEJ process that does not exist when they begin execution. A single MERGEJ process is created as a result of the first port operation on the port established from the object type linked to the MERGEJ process image, regardless of which process is first to use its output port. Thus, the creation of the MERGEJ process is entirely invisible to the SCAN and EXHSEL processes and is independent of which of the SCAN and EXHSEL processes is first to send a tuple.

7. SUMMARY

We have presented the use of two concepts, type ports and cooperation classes, in the establishment of interprocess communication paths and in the creation of processes in a port-based system. We have demonstrated that two process initiation protocols (creative and conservative) and the ability to pass port use privileges allow the construction of arbitrary process interconnection topologies, without process use of the system's process identifiers.

The purpose of the Gutenberg interprocess communication facility is to allow all interprocess communication to be organized as requests for and performance of operations on abstract data types. One of the problems faced in implementing such a paradigm, and indeed in any system where objects are shared, is that of protecting remote objects from unauthorized access. Though the two process initiation protocols are sufficient for implementing object sharing, the concept of cooperation class was

introduced as a mechanism the kernel uses to provide object protection, and to simplify the problem of designing abstract data type managers. A second use of cooperation classes was also demonstrated by presenting an example of a data-driven relational query execution technique. It was argued that cooperation classes represent a new concept useful in constructing systems of communicating processes. Further research in the use of cooperation classes for protecting objects and in concurrency control of database transactions is in progress.

8. REFERENCES

[ANCI83] Ancilotti, P., Boari, M., Lijtmaer, N., "Language Features for Access Control," IEEE Transactions on Software Engineering, vol. SE-9, no. 1, Jan. 1983.

[ANDR81] Andrews, G., "Synchronizing Resources," ACM Transactions on Programming Languages and Systems, vol. 3, no. 4, October, 1981.

[BALL76] Ball, J. E., Feldman, J., Low, J., Rashid, R., Rovnet, P., "RIG, Rochester's Intelligent Gateway: System Overview," IEEE TSE, 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, 6th ACM Symposium on Operating System Principles, November, 1977.

[BRIN78] Brinch Hanson, P., "Distributed Processes: A Concurrent Programming Concept", Communications of the ACM, vol 21, no. 11, November, 1978.

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

[CHER79] Cheriton, D. R., Malcolm, M. A., Melen, L. S., Sager, G. G., "Thoth, A Portable Real-Time Operating System," CACM, vol. 22, no. 2, 1979.

[COHE75] Cohen, E., Jefferson, D., "Protection in the HYDRA Operating System," Proceedings, 5th ACM Symposium on Operating System Principles, 1975.

[COOK80] Cook, R., "*MOD - A Language for Distributed Programming," IEEE TSE, vol. SE-6, no. 6, November, 1980.

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

[DATE83] Date, C. J., *An Introduction to Database Systems. vol II*, Addison-Wesley Publishing Company, Reading, Ma. 1983.

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

[DERE76] DeRemer, F., Kron, H., "Programming-in-the-Large Versus Programming-in-the-Small," IEEE Transactions on Software Engineering, vol. SE-2, no. 2, June 1976.

[DOD83] "Reference Manual for the Ada Programming Language," (ANSI/MIL-STD-1815A), U.S. Department of Defense, Washington, D.C., January, 1983.

[ENGL74] England, D., "Capability Concept Mechanism and Structure in System 250," Proceedings, International Workshop on Protection in Operating Systems, Aug. 1974.

[FELD79] Feldman, J., "High Level Programming for Distributed Computing," Communications of the ACM, vol. 22, no. 6, June, 1979.

[GELE82] Gelernter, D., Bernstein, A. J., "Distributed Communication via Global Buffer," Proceedings, 1st ACM Symposium on Principles of Distributed Computing," August, 1982.

[GENT81] Gentleman, W. M., "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept," Software - Practice and Experience, vol. 11, 1981.

[GOOD79] Good, D. I., Cohen, R. M., Keaton-Williams, J., "Principles of Proving Concurrent Programs in Gypsy," Proceedings, 6th ACM Symposium on Principles of Programming Languages," January, 1979.

[HOAR78] Hoare, C. A. R., "Communicating Sequential Processes," CACM, vol. 21, no. 8, August, 1978.

[KAHN81] Kahn, K. C., et. al., "iMAX: A Multiprocessor Operating System for an Object-Based Computer," Proceedings, 8th ACM Symposium on Operating System Principles, December, 1981.

[HOAR78] Hoare, C. A. R., "Communicating Sequential Processes," CACM, vol. 21, no. 8, August, 1978.

[KAHN81] Kahn, K., Corwin, W., Dennis, T., D'Hooge, H., Hubka, D., Hutchins, L., Montague, J., Pollack, F., "iMAX: A Multiprocessor Operating System for an Object-Based Computer," Proceedings, 8th ACM Symposium on Operating System Principles, Dec. 1981.

[KIEB78] Kieburtz, B., Silberschatz, A., "Capability Managers," IEEE Transactions on Software Engineering, vol. SE-4, no. 6, Nov. 1978.

[LANT80] Lantz, K., "Uniform Interfaces for Distributed Systems," University of Rochester Tech Report TR63, May, 1980.

[LISK79] Liskov, Barbara, "Primitives for Distributed Computing", Proceedings, 7th ACM Symposium of Operating System Principles, December, 1979.

[MAO80] Mao, T. W. Yeh, R., "Communication Port: A Language Concept for Concurrent Programming," IEEE TSE, vol. SE-6, no. 2, March, 1980.

[MCGR79] McGraw, J., and Andrews, G., "Access Control in Parallel Programs," IEEE Transactions on Software Engineering, vol. SE-5, no. 1, Jan. 1976.

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

[NSW76] NSW Protocol Committee, "MSG: The Interprocess Communication Facility for the National Software Works," Bolt, Beranek, and Newman Report 3237, January, 1976.

[RAMA83] Ramamritham, K., Briggs, D., Stemple, D., Vinter, S., "Privilege Transfer and Revocation in a Port-based System," University of Massachusetts COINS Technical Report, October, 1983.

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

[RASH81] Rashid, R., Robertson, G., "Accent: A Communication Oriented Network Operating System Kernel," Carnegie-Mellon University 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.

[SPEC82] Spector, A., "Performing Remote Operations Efficiently on a Local Computer Network," CACM, vol. 24, no. 4, April, 1982.

[STEM83] Stemple, D., Ramamritham, K., Vinter, S., "Operating System Support for Abstract Database Types," Proceedings, 2nd International Conference on Databases, September, 1983.

[STEM84] Stemple, D., Vinter, S., and Ramamritham, K., "Module Interconnection Control: Support for Secure Systems," submitted for publication.

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

[TICH79] Tichy, W., "Software Development Control Based on Module Interconnection," Proceedings, Fourth International Conference on Software Engineering, Sept. 1979.

[VINT83] Vinter, S., Ramamritham, K., Stemple, D., "Protecting Objects Through the Use of Ports," Proceedings, Pheonix Conference on Computers and Communication, March, 1983.

[VOYD80] Voydock, V., "Features on Network Interprocess Communication Protocols," BBN Report 4489, September, 1980.

[WALD72] Walden, D. C., "A System for Interprocess Communication in a Resource Sharing Computer Network," CACM, vol. 15, no. 4, April, 1972.