The Gutenberg Operating System Kernel
[1]

Panayiotis K. Chrysanthis
Krithivasan Ramamritham
David W. Stemple
Stephen T. Vinter [2]

COINS Technical Report 86–06
February 1986

[1]This material is based upon work supported in part by the National Science Foundation under grants DCR-8403097, and DCR-8500332.

[2]Current address BBN Laboratories, Cambridge, Mass.

Abstract

The Gutenberg system is a port-based, object-oriented operating system kernel designed to facilitate the design and structuring of processes in distributed systems. This is achieved by providing primitives for structuring processes through controlled process interconnections and through controlled access to shared resources. Only shared resources are viewed as protected objects and hence, all shared resources can be accessed only by specific operations defined on them. Processes communicate with each other and access protected objects through the use of ports. Each port is associated with an abstract data type operation and can be created by a process only if the process possesses the privilege to execute that operation. Privileges are represented by one or more capabilities. Thus, access control to shared resources is achieved by controlling acquisition of capabilities to create ports by processes.

Capabilities to create ports for requesting operations are contained in the *capability directory* which is a kernel object, i.e., is maintained and manipulated only by the kernel. The purpose of the capability directory is to restrict the use of capabilities in a consistent and orderly way. At any time, each process is associated with a single subdirectory of the capability directory designated as its *active directory*. The protection domain of a process is defined by the set of stable capabilities in its active directory and of transient capabilities in its capability list. Unlike stable capabilities, transient capabilities exist only for the lifetime of the process that owns them.

This paper describes the design philosophy and the structure of the Gutenberg kernel. First, it discusses the principles and motivations behind the Gutenberg design in some depth. Then, it presents the structure, contents and operations of kernel objects and discusses their use in structuring acess to protected user-defined objects. The management of processes is also discussed. It concludes with an example of controlled interprocess communication using the Gutenberg mechanisms.

1 INTRODUCTION

In the design of distributed systems, the protection of resources in the system, in spite of their shared use, is an important issue. The protection problem is exacerbated in distributed systems by three factors:

1. Processes that use a resource and the resource itself may be placed on separate nodes in the system. This implies that authorization checks may involve communication overheads.

2. Communication links that connect the nodes themselves are shared resources and hence have to be protected. This introduces further overheads.

3. The protection facility itself needs to be distributed to ensure its continuous, reliable performance. This requires the cooperative interaction of the nodes of the system in which control is decentralized.

These point to the need for an efficient mechanism that achieves protection of shared resources in a distributed system with minimal overheads.

This paper deals with the design of such a mechanism, the distributed Gutenberg kernel [**Stemple et al. 83, 85, 86, Ramamritham et al. 83, 85, 86**]. The Gutenberg Operating System Kernel currently being developed at the University of Massachusetts takes a unique approach to protection. Three features of Gutenberg facilitate the realization of a coherent decentralized protection scheme and should contribute to its better performance compared to previous protection schemes:

- the adoption of *port-based protection*, whereby control of access to shared resources is achieved efficiently by controlling the creation of ports (communication links) connecting processes using a resource to the process managing the resource;

- the adoption of *non-uniform object-orientation*, whereby only shared resources need be structured as objects, i.e., entities that can be accessed only via specific operations defined on them; and

- the use of a *capability directory*, which is a distributed repository of persistent protection information and is maintained and manipulated only by the kernel.

The primitives provided for the control of interprocess communication and access to shared objects support the design and structuring of distributed computations. Typically, sharing of resources in protected systems is achieved through the distribution of access rights, i.e. privileges, to subjects. Information about accessors' rights may be stored with the objects being shared, as in the case of access lists [**Saltzer 74**] or with the accessor, as in the case of capabilities [**Dennis et al. 66**]. (Capabilities are unforgeable tickets that authorize their possessor to access a named object via a subset of the operation defined by the object's type.) In either case, protection is achieved by checking that the accessor has previously been granted the right to perform the operation.

The protection scheme used in Cambridge File Server (CFS) [**Dion 80**] is typical of capability-based protection schemes. Along with an access request, a process sends the appropriate capability to the file server via a communication link. If the capability is valid, the server satisfies the request. This scheme requires protected communication channels because they are used to transmit capabilities. In such systems, the steps involved in accessing a remote protected resource are: (1) A process on a node in the distributed system presents a capability for the appropriate communication channel; (2) Via the communication channel, the process sends a capability for the resource and the request to the remote process that manages the resource; (3) The resource manager process authenticates the capability provided by the user process; (4) The resource manager process responds to the request. Thus, one local capability check, one remote capability check, and a two-way communication are needed to access a remote protected object. This represents a significant overhead. In the next section we propose an approach to protection which incurs less overheads than a typical capability-based scheme, but affords similar benefits.

Proposed Approach to Protection in Distributed Systems. The principle underlying our *port-based* approach to protection can be summarized as follows: *Permit a user process to create a port (a communication channel) to the process managing a remote protected resource only if the user process has the right to access the resource.* If such checks are placed on port creation, then the only check that needs to be made when a process requests access to a remote resource via a port is whether the port belongs to that process and whether its intended use is in the manner specified at the time the channel was created. In Gutenberg, this required check is done at the node in which the user process resides; it is a local check.

All shared resources in Gutenberg are structured as objects and hence can be accessed only via specific operations defined on them; a process may access a shared resource only if it has access to a port that is specifically associated with a specified operation on a resource of a specified type, and is connected to the resource's manager. This association of operations on a resource (i.e. object) and ports gives Gutenberg an *object-orientation*. However, in Gutenberg this object-orientation is *non-uniform* in that it is required only at the level of resources *shared* between processes.

We believe that conventional mechanisms for protecting unshared data (e.g., local variables) are adequate and desirable, from both a downward compatibility and an efficiency viewpoint. Conventional memory protection mechanisms (e.g. page tables) ensure that the local data of processes are not accessible to other processes. Gutenberg depends on programming languages to provide static type-checking for self-protection. Thus, although local data *are not* necessarily object-oriented (depending on the programming language), shared data *are* required to be object-oriented. As a result of this non-uniform object orientation, Gutenberg will not suffer from the performance problems that arise from dynamic access authorization for all accesses to objects, as with *uniform object-orientation*, e.g. Hydra [**Wulf et al. 74**] and iMAX [Cox et al. **81**].

Gutenberg controls the creation and use of ports through the use of *capabilities*. The control of capabilities is a central theme in Gutenberg. Our approach to this control can be summarized as follows: *Capabilities for potentially sharable objects are maintained in a physically distributed but logically unified structure to ensure that they are accessed in a controlled and reliable manner.* Being physically distributed, this structure, called the *capability directory*, can be designed to ensure availability and reliable access despite communication and node failures. However, since it is logically unified, i.e., appears as a single globally addressable entity, the physical distribution will be transparent to its accessors.

The purpose of the capability directory is to restrict the use of capabilities in a consistent and orderly way. At any time, each process is associated with a single subdirectory of the capability directory designated as its *active directory*. The protection domain of a process is defined by the set of stable capabilities in its active directory and of transient capabilities in its capability list. Unlike stable capabilities, transient capabilities exist only for the lifetime of the process that owns them.

In addition to the performance enhancements made possible by the use of port-based protection and non-uniform object-orientation, use of the capability directory, which is maintained and manipulated only by the kernel, contributes to the efficacy of protection in Gutenberg.

Communication and Protection in Related Systems. A few systems provide port-based communication facilities using functional addressing [**Stemple et al. 86**], but none tie protection so closely to communication as does Gutenberg. The Accent system is port-based and supports asynchronous communication with process transparency [**Rashid et al. 81**]. Communication in Gutenberg is also similar to the mechanisms used in Intel iAPX-432 [**Cox et al. 81**], DEMOS [**Baskett et al. 77**] and NIL [**Strom et al. 83**]. In all these systems, apart from NIL, even though a communication link could be typed, thus restricting its use, they do not support the concept of restricting access to shared objects by restricting the creation of communication channels, as in Gutenberg.

A distinguishing feature of procedure invocation in systems such as Hydra (and the iAPX 432) is the creation of a local name space (LNS) with each (protected) procedure invocation. This LNS defines the protection domains for the execution of the procedure. It is composed of the union of the capabilities for actual parameters passed in the invocation and the capabilities possessed by the inactive procedure itself. The creation of local name spaces greatly increases the cost of a procedure call. Since accessing a shared object involves communication between processes in Gutenberg, local name spaces in these systems correspond to process name spaces and appropriated process protection domains in Gutenberg, and the creation of a local name space corresponds to a context switch in Gutenberg. This approach has been taken in other systems, such as with the NIL programming language [**Strom et al. 83**]).

We believe that the Gutenberg approach has two benefits. First, by encapsulating objects in active entities (processes) instead of passive entities (procedures), the requestor of an operation on an object can execute concurrently with the execution of the operation. When accessing a remote object using asynchronous communication, a context switch can be avoided altogether. The exploitation of concurrency was a major motivation for the use of resource managers in Gutenberg and the selection of communication primitives. And second, processes in Gutenberg are able to change their protection domain by changing their active directories, i.e., their points of reference within the capability directory. This change is expected be very inexpensive compared with the creation of an LNS.

As mentioned earlier, a unique feature of Gutenberg is the capability directory, which contains stable capabilities in a unified structure controlled by the kernel. Other systems, such as Hydra, iAPX 432, and CAL [**Lampson et al. 76**] allow capabilities to be stored in inactive objects (i.e., data structures, as opposed to processes) that are not kernel objects. The problem of how to allow such objects to be permanently stored in secondary memory is noted in [**Lampson et al. 76**]. Having a unified structure for stable capabilities that is separate from user-managed data facilitates their management by the kernel and their use by application processes.

2 PRINCIPLES UNDERLYING THE GUTENBERG KERNEL

The Gutenberg system evolved from a series of design decisions concerning the nature of protection and communication in the system. We discuss these now.

Limit the responsibilities of the kernel. Operating systems that support the definition and protection of arbitrary objects traditionally have had performance problems because of the maintenance of protection domains and the overhead of dynamic access checks. We hoped to restrict the overheads of the kernel by taking a *non-uniform object orientation,* in which only resources shared by different processes need to be structured as objects at the operating system level. An underlying assumption in Gutenberg is that the granularity of objects is medium to large, a size adequate to amortize the cost of the interprocess communication required to access the object.

Programming language independence. There should be no requirement that a specific programming language or that only object-oriented languages run under the Gutenberg system kernel. In fact, non-object-oriented languages can achieve an object oriented view through the use of kernel primitives [**Stemple et al. 83**]. The system is designed to support applications implemented in any programming language.

Resources are directly manipulated only by their managers. Managers are processes that synchronize the operations on an object (i.e., a shared resource). A process managing an object is the only subject able to directly manipulate the object. The process performs the operations, in part, by invoking kernel primitives for manipulating kernel objects.

Object sharing is via interprocess communication. Processes do not share address spaces. They can interact only through interprocess communication using explicit message passing. An operation can be performed on the object only as a result of a request from another process via interprocess communication. Processes are provided with communication primitives allowing both synchronous and asynchronous communication.

Interprocess communication connections are established using functional addressing [**Vinter et al. 83, Stemple et al. 86**]. Processes use communication channels called *ports* to request operations on objects. A port can be used to request an operation only if it was created for that purpose. Ports are created by indicating the function of the port (viz., to request an operation), not by identifying a particular process.

The kernel controls access to shared objects by controlling interprocess communication. An operation may be requested only by transmitting the operation request over a port to the object's manager. By limiting the use of ports and constraining their use to request a specific operation on a specific object, access to the object is controlled.

Details of the implementation of the communication mechanism are hidden from the processes using it. Ports are themselves objects with a small set of operations defined on them. They are managed by the kernel. The representation of ports and details of message transmission and reception are hidden from communicating processes.

Privileges persist in a single kernel-managed structure. Gutenberg recognizes that privileges need to persist in the system independent of the execution of processes. Rather than allowing privileges to be placed on secondary storage in user objects (hidden from the view of the kernel), privileges that are not dependent on the existence of a process are stored in a kernel managed structure called the *capability directory.* This directory is shared by the processes in the system. Transient capabilities of a process, i.e., capabilities that exist only as long as the process is active, are kept in the process' *c-list.*

The above design principles made it possible to use Gutenberg as the basis for a distributed operating system for the following reasons. The use of resource managers is a common approach to structuring software in distributed systems. Second, the logical separation of process address spaces in Gutenberg corresponds to the physical realization of processes in a distributed system. Third, the close association of communication and protection contributes to decentralized access authorization. And last, the intent to increase concurrency by using asynchronous communication is appropriate in a distributed system, where communication delays increase overheads and there is great opportunity for parallelism.

3 GUTENBERG KERNEL OBJECTS

In the non-uniform object model adopted in Gutenberg, the kernel primitives invoked by processes to manipulate kernel objects are viewed as abstract data type operations on kernel objects. Thus, the kernel is viewed as an abstract data type manager of kernel objects implemented within its address space. This view is common to both the kernel and the processes that invoke the primitives; just as user object managers cannot view the internals of the kernel, the kernel cannot view the internals of the object managers. This implies that control of a user object is strictly within the jurisdiction of the user object manager. This uniform view is expected to simplify and improve the correctness of both the kernel implementation as well as the structuring of the other processes within the system, since it leads to an inherently modular system which supports information hiding and controlled interface. The kernel consists of individual modules that manage each of the kernel objects. There are four types of kernel objects: *processes*, *ports*, *capability directory* and *transient capabilities*.

3.1 Processes

A process is an independently schedulable unit of computation, with access to protected objects. Each process is represented by a unique *process control block*, abbreviated *PCB*, which resides within the kernel address space. Processes can be classified into *system processes* and *user processes*. System processes together with the kernel form the operating system, which is viewed as a cooperative set of managers of the system objects. Typically all system objects are protected. An example of a system process is the file manager. User processes are not part of the operating system, but manage shared, user-defined objects. This is achieved in the same way as with system processes, by accepting operation requests, executing the operations and returning the results to the requesting processes. To help distinguish between requests to the kernel and request to access protected objects, user-defined or system-defined, we say that processes execute *kernel primitives* to manipulate kernel objects, and request *operations* to manipulate protected objects.

It becomes clear from the above that user processes interact with both system processes and with other user processes. In Gutenberg, processes can only communicate through explicit message exchanges over communication channels called *ports*. As a result, processes do not *share address spaces*, eliminating the need for synchronization in memory access and generalizing the process interactions in a distributed system. Other systems, e.g. NIL [**Strom et al. 83**] and Accent [**Rashid et al. 81**], adopt a similar model for processes in their system.

Port Type	Client Primitives	Server Primitives
S	SEND REVOKE	RECEIVE REFUSE
R	RECEIVE	SEND REFUSE
SR	SEND-RECEIVE REVOKE	GETDETAILS SEND REFUSE

Figure 1: Port primitives used by clients and servers for each port type

3.2 Ports

A port is a kernel object that processes manipulate by invoking kernel primitives. As described above, a port is a communication channel between a pair of processes in one-to-one topology. Thus, a port connects just one pair of processes at a time. One process has the privilege to place messages on the port, which behaves as a queue of messages awaiting delivery. The other has the privilege to remove messages. This communication can be either synchronous or asynchronous.

The basic interprocess communication of the Gutenberg system is based on the client/server model, in which the creator of a port, called the *client*, communicates with the port server, the manager of some shared object, for the purpose of requesting an operation on the object. A port is established with *functional addressing*; a client creates a port by naming the service it would like to request using the port rather than by identifying the server process. As a result, the server process does not have to be in existence prior to the creation of the port. The advantage of this strategy is that it allows the dynamic creation of server processes. In fact, in Gutenberg, process creation and destruction are byproducts of port operations. There are no primitives for process creation and destruction: *processes are hidden from the programmers, the lowest level of abstraction being the level of operations on the ports.*

The only way a process can request an operation on a shared object is to create a port and execute a kernel primitive on that port. [3] The possible kernel primitives on ports that a client is entitled to, include SEND, RECEIVE, and SEND-RECEIVE (to receive the result of an operation based on the parameters sent).

In order to restrict the use of ports to the functionality for which they have been created, a port is typed as either a Send, Receive, or Send-receive port. This typing specifies the

[3]One implication of port creation requests being satisfied by the kernel in Gutenberg is that once a port is created, it is not possible to automatically control the number of times a specific port is used. Such a facility must be explicitly programmed into servers in Gutenberg.

directionality of the port and the kernel primitives used by the client to transmit messages through it. Consequently, it specifies the kernel primitives that the server of the port may use. Figure 1 shows the port primitives used by clients and servers for each port type. Send and Receive ports are unidirectional. Send-receive ports are bidirectional, allowing the port's client to send a message and receive a response from the port's server.

Port typing also determines the format of messages that may be placed in the port and the *object operation* associated with this, which identifies the operation that will be requested via the port. Each port is represented by a unique *channel control block*, abbreviated *CCB*, which resides within the kernel address space. CCB contains the port type along with information about the status of the client and server processes and the owner of the port. Ownership represents the privilege to destroy the port. The creator of the port becomes the initial owner of the port. As part of the sharing mechanism supported by the Gutenberg system, a process may transfer part of its privileges, including port privileges, to another, over ports.

Here is a short summary of the purposes of kernel primitives on ports. (Details concerning the primitives may be found in [**Ramamritham et al. 83**]).

CREATE-PORT (only a client primitive) creates a port of a specific type. The type is specified via a parameter to the call.

DESTROY-PORT (only a client primitive) destroys a port. The caller must be the owner of the port. The port-id is specified via a parameter to the call.

SEND puts a message on a port. The system has two kinds of SEND primitives: acknowledge-SEND and no-acknowledge-SEND. If the SEND is an acknowledge-SEND, the sending process is informed when its correspondent over the port receives the message, and can choose to block until the receipt of the acknowledgement.

RECEIVE requests the next message from the port. The caller elects via a parameter to the call, to either block, if there is no message on the port, or execute concurrently with the servicing of the request.

SEND-RECEIVE (only a client primitive) puts information, termed *request details*, on a port for the server to use in satisfying the request. When the server responds to the request by executing a SEND, the server's reply is returned to the client as in RECEIVE. The caller may block until the server replies, or execute concurrently with the servicing of the request.

ACCEPT-REQUEST (only a server primitive) is used to obtain access to newly created ports and to query a set of existing ports to see if new messages have arrived. The caller may block until the kernel replies, or execute concurrently with the servicing of the request.

GETDETAILS gets request details from a port. The caller (the port's server) may block if there is no pending SEND-RECEIVE, and thus no request details, on the port, or it may execute concurrently with the satisfaction of its request.

REFUSE rejects a client's request for service as unsatisfiable and notifies the requester by setting a status.

REVOKE revokes privileges sent as part of request details by a SEND-RECEIVE or in a SEND message up to receipt of the message [4].

The choice of these primitives during system design was based on the desire to keep their number and complexity to a minimum while providing users a set of primitives for building systems of communicating processes in arbitrary topologies (see [**Vinter et al. 83**]) with reasonable ease. Thus, we have added to the basic SEND and RECEIVE primitives the bidirectional SEND-RECEIVE and its receiving reciprocal GETDETAILS in order to allow such functions as reading a record with a given key (the key being sent as request details) or a remote procedure call (the procedure's parameters being sent as request details) to be implemented by a single primitive. However, it should be noted that the asynchronous mode makes the SEND-RECEIVE operation more robust and flexible than a remote procedure call semantics.

3.3 Capability Directory

Gutenberg controls the creation and use of ports through the use of capabilities. All capabilities for accessing potentially sharable objects are maintained in a logically unified structure called the capability directory. Thus, the capability directory expresses all potential process interconnections in the system. This is similar to the UNIX file directory [**Ritchie et al. 74**] which provides uniform treatment of files, devices and interprocess communication. The capability directory is a *stable* structure in that its existence does not depend on the existence of any process. It is also *shared* since more that one process may concurrently access the same segment of the directory. It should be noted that no portion of the directory is owned by any process at any time.

3.3.1 Capability Directory Nodes

Capabilities within the capability directory are further organized into groups called the *capability directory nodes*, abbreviated *cd-nodes*. They are identified by both a system-wide unique name created by and visible only to the kernel, and by user-specified names. In general, cd-nodes contain other information along with capabilities. Cd-nodes are linked to

[4] A scheme for revoking transferred capabilities anytime after the transfer is discussed in [**Ramamritham et al. 86**]

other cd-nodes through capabilities. The same cd-node may be linked to several cd-nodes under possibly different user-specified names. All capabilities point to a cd-node have equal status. That is, cd-nodes are unique and are not contained within other cd-nodes. A cd-node exists independently of any other cd-node and disappears along with the last capability link to it, if it is not explicitly destroyed. In this way the capability directory is structured as a graph in which nodes (each node corresponds to a cd-node) are connected by edges corresponding to capabilities. Figure 2 shows an example of capability directory segment which implements the mail facility in a system. The capability directory may contain two kinds of cd-nodes: *subdirectories* and *manager definitions.*

A subdirectory is a list of capabilities (see figure 3). It is merely an organizational unit of the capability directory, similar to a file directory in a file system. At any time, every process in the system is associated with a single subdirectory in the capability directory designated as its *active directory.* The active directory of a process is the set of capabilities from the capability directory that a process may use or exercise.

The active directory of a process is one component of a process protection domain. The other component is its current set of transient capabilities; this is discussed later. A process may dynamically switch from one protection domain to another by changing to a new active directory or changing the contents of its current active directory, if it has the privilege to do so.

Manager definitions constitute one of the novel features of the Gutenberg system. All processes in the system are instantiated from manager definitions (figure 4). Thus, a manager definition provides information necessary for instantiating the manager process, as, for example, a capability for the file containing the executable image (object module) of the process, and the initiation protocol (see section 4.2) for determining the manner in which ports are connected to manager processes. It also includes a capability for a subdirectory containing the privileges that all processes instantiated from the manager definition will initially possess.

One of the components of a manager definition node is a set of *port descriptors.* This set corresponds to the set of operations defined within the manager. Each port description contains a *generic operation name* (a name specified by the user at manager creation time), and the type of port through which a user requests this operation. It should be noted that manager definitions, unlike subdirectories, never define a part of the protection domain of any process.

Capabilities in cd-nodes inherit all the properties of the capability directory in that they are stable, sharable but not owned by processes. These capabilities are called *stable* capabilities. Stable capabilities can only reside in the capability directory.

Capabilities in Gutenberg consist of three parts: a specific kernel primitive, a list of parameters for the primitive, and a list of primitives that can be used to manipulate the

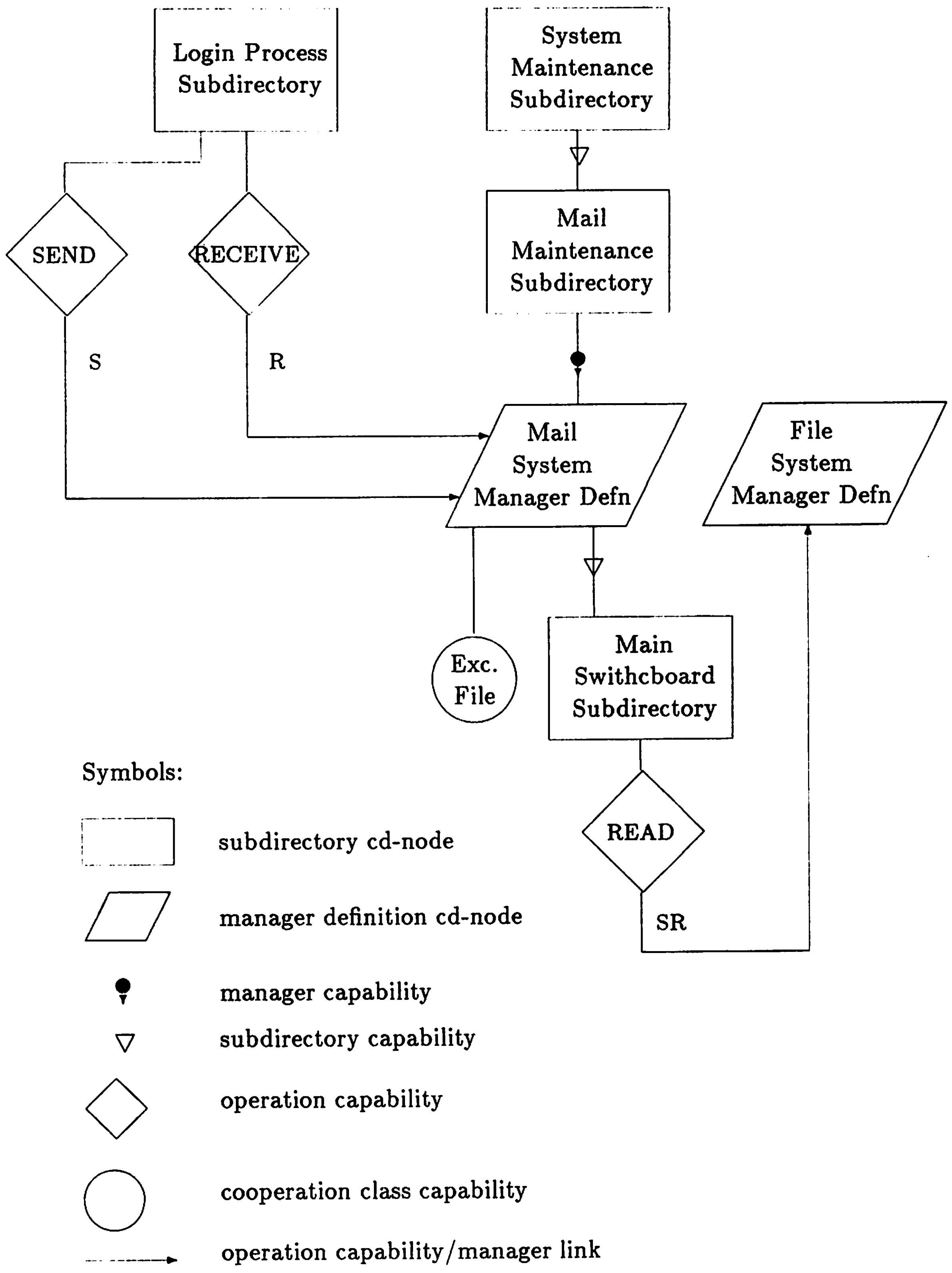

Login Process Subdirectory
System Maintenance Subdirectory
SEND
RECEIVE
Mail Maintenance Subdirectory
S
R
Mail System Manager Defn
File System Manager Defn
Exc. File
Main Swithcboard Subdirectory
READ
SR
Symbols:
subdirectory cd-node
manager definition cd-node
manager capability
subdirectory capability
operation capability
cooperation class capability
operation capability/manager link

Figure 2: Example of Capability Directory Segment

Subdirectory cd-node Contains a set of capabilities, and has the following attributes:

> **subdirectory id** system-wide unique identifier of the subdirectory. (Cannot be modified through the **Modify** primitive - it is generated by the kernel upon creation of the cd-node.)
>
> **use count** the number of stable and transient subdirectory capabilities (including those in manager definition cd-nodes) pointing to this cd-node. (Cannot be modified through the **Modify** primitive - it is maintained by the kernel.)
>
> **active directory count** the number of processes having this cd-node as their active subdirectory. (Cannot be modified through the **Modify** primitive - it is maintained by the kernel.)

Figure 3: Contents of Subdirectory cd-node

capability itself, which are called *capcaps*, for capabilities on a capability.

A capability permits a process which possesses it to invoke the specific kernel primitive it contains. This primitive is also called *primary* kernel primitive in order to distinguish it from the other kernel primitives that manipulate the capability itself.

The capcaps determine how the capability may be modified and used (figure 5). Capcaps include the privilege to *transfer* (to another process), *copy*, *register* (make stable), *hold* (make transient), *merge* (with other mergeable capabilities), *view*, and *modify* the capability. Each capcap may be active, in which case the corresponding kernel primitive may be invoked for the capability, or inactive, in which case the corresponding kernel primitive cannot be invoked on the capability. Not every capcap makes sense for each type of capability. When we discuss the specific capabilities next we point out the capcaps that are applicable.

The parameter list may include names of cd-nodes as well as other capabilities (most notably, the cooperation class capability which is discussed later). A parameter may have one of three different kinds of values: a value, a list of values which may not be added to, or ANY. ANY indicates that the parameter is not specified and may be replaced by any value or list of values. The parameter list associated with a capability depends on the type of the capability. Hence, we now discuss the four different capability types.

3.3.2 Types of Capabilities

There are four different types of capabilities that may be stored in the capability directory: *operation*, *subdirectory*, *manager definition*, and *cooperation class capabilities*.

An operation capability (figure 6) represents the privilege to create a port for use in

Manager Definition Node Provides information necessary for instantiating a manager process. The manager definition cd-node has eight attributes. *Value* describes legal value(s) that the attribute may have; *Default* is the value the attribute is set to when the cd-node is created. The attributes are:

manager id system-wide unique identifier of the manager cd-node. (Cannot be modified through the **Modify** primitive - it is generated by the kernel upon creation of the cd-node.)

initial active directory a subdirectory capability pointing to the subdirectory cd-node that will become the active directory of the manager process when it is initiated.
Value: any subdirectory capability. (If no value is specified, at the time of instatiation of a process from this manager definition, the kernel creates an empty subdirectory to serve as the active directory of this process.)

initial process image a file capability (represented as a cooperation class capability) containing the object code to be executed when the process is initiated.
Value: any cooperation class.

manager dependency indicates whether the existence of the process is dependent on the existence of ports connected to the manager process.
Values: independent (I) or dependent (D).

initiation protocol indicates when a new manager process is created or an existing one is connected to, when a port to the manager is created.
Values: creative (CR), conservative (CO), or class conservative (CC).

port descriptors specifies the list of (port type, generic operation name) pairs for operation capabilities that may be linked to this manager cd-node. The port type, except for the link type, specifies the format of the arguments that may be passed over the port as part of a message or request-details.
Port type values: send (S), receive (R), send-receive (SR). Argument format value: privilege part of a message, privilege part of request-details, non-privilege part of a message, non-privilege part of request-details, or ANY. Operation name values: a name or ANY. Default: S:ANY:ANY, R:ANY:ANY, SR:ANY:ANY (for both message and request-details).

manager use count the number of manager capabilities pointing to this cd-node. (Cannot be modified through the **Modify** primitive - it is maintained by the kernel.)

operation use count the number of operation capabilities linked to this cd-node. (Cannot be modified through the **Modify** primitive - it is maintained by the kernel.)

Figure 4: Contents of Manager Definition cd-node

COPY the capability may be copied (applicable only to *nonexclusive* capabilities);

TRANSFER the capability may be transferred; (if the capability is nonexclusive then a copy of the capability is transferred, otherwise the capability itself is transferred);

MERGE the capability may be associated with another compatible capability in order to augment its capcaps and/or rights.

REGISTER the transient capability may be registered in the capability directory;

REMOVE the stable capability may be removed from the capability directory;

HOLD the stable capability may be made transient;

VIEW-NODE the contents of a cd-node pointed to by the capability may be examined;

MODIFY-NODE the contents of the cd-node pointed to by the capability may be modified.

DESTROY-NODE the cd-node pointed to by the capability may be destroyed;

VIEW-CAP the contents of the capability may be examined;

MODIFY-CAP contents of the capability (excluding the capcaps) may be modified;

MODIFY-CAPCAP the capability's capcaps may be modified;

Figure 5: List of capcaps and their meaning in the case they are active

Operation Capability Provides the privilege to create ports. Thus, the primitive associated with this capability is **Create-port**. Operation capabilities have six attributes:

operation name user-specified name of the operation. This name becomes the object operation of created ports. This name also serves to identify the operation capability.
Value: a name; it must be specified upon creation of the capability.

generic operation name name of the corresponding operation defined in the manager definition cd-node. This name can be the same as or different from the operation name. (Cannot be modified through the **Modify** primitive - it is generated by the kernel upon creation of the capability.)

port type specifies the port type for this operation that may be linked to the corresponding manager definition cd-node. The port type can either be Send (S), Receive (R), or Send-receive (SR). (Cannot be modified through the **Modify** primitive - it is generated by the kernel upon creation of the capability.)

message format specifies the arguments that may be passed over the port as part of the message, and the request-details in case of send-receive port type. (Cannot be modified through the **Modify** primitive - it is generated by the kernel upon creation of the capability.)

cooperation class(es) restrict how the port will be connected to a manager process.
Value: cooperation class capability or ANY. Default: ANY; in this case, according to the initiation protocol, cooperation class is required to be specified upon invocation of the create-port primitive.

manager id a pointer, global name, to the manager to which this operation capability is linked. (cannot be modified through the **Modify** primitive - it is set by the kernel upon creation of the capability.)

The capcaps of the operation capability are: COPY, TRANSFER, REGISTER, REMOVE, HOLD, MERGE, MODIFY-CAP, MODIFY-CAPCAP, and VIEW-CAP.
Value: each is active or inactive. Default: all active.

Figure 6: Contents of an Operation Capability

Manager Definition Capability Provides the privilege to create operation capabilities. Thus, the primitive associated with this capability type is **Create-operation**. The manager definition capability has the following attributes:

manager definition name user-specified name used to identify the manager definition cd-node to which the created operation capabilities are linked to. This name also serves to identify the manager definition capability.
Value: a name; it must be specified upon creation of the manager definition.

cooperation class(es) restrict who may create operation capabilities linked to the definition manager. When exercising the create-operation privilege, a process must possess one of the specified cooperation class capabilities or else the primitive is illegal.
Values: cooperation class identifier(s) or ANY. Default: ANY.

manager id a pointer, global name, to the manager definition cd-node corresponding to this capability. (cannot be modified through the **Modify** primitive - it is set by the kernel upon creation of the capability.)

The capcaps of the manager definition capability are: COPY, TRANSFER, HOLD, REGISTER, REMOVE, DESTROY-NODE, MERGE, MODIFY-CAP, MODIFY-NODE, MODIFY-CAPCAP, VIEW-CAP, and VIEW-NODE.
Value: each is active or inactive. Default: they are all active.

Figure 7: Contents of a Manager Definition Capability

requesting a particular operation on a given user-defined object type. Thus, the primary kernel primitive of the operation capability is **Create-port**. One parameter of the operation capability is the operation name, which becomes the operation requested via ports created from this capability. This name also serves to identify the operation capability in the subdirectory in which the capability is contained. Another parameter of the capability is the name of a manager definition cd-node in the capability directory that the operation capability is *linked* to in the capability directory. This manager definition is used by the kernel to determine whether a newly created port is to be connected to a new server process instantiated from the manager definition or to an already existing one. It is also used by the kernel in conjunction with a third parameter, the generic operation name, to determine whether the requested operation is supported by the manager; this is checked by examining whether the operation generic name is part of the port descriptors in the manager definition. If the requested operation is no longer defined in the manager definition, the invocation of the create-port primitive is not performed and the status is returned.

The primary kernel primitive in the manager definition capability (figure 7) is **Create-operation**, which is used to create operation capabilities, linked to the manager definition

Subdirectory Capability Provides the privilege to change the process' active directory. Thus, the primitive associated with this capability is **change-directory**. The attributes are:

subdirectory name user-specified name of the subdirectory cd-node which becomes process' active directory when the **change-directory** privilege is exercised. This name also serves to identify the subdirectory capability.
Value: a name. Default: none; must be specified upon creation of the subdirectory.

cooperation class(es) used to restrict who may make the subdirectory active. When exercising the **Change-directory** privilege, a process must possess one of specified cooperation class capabilities or else the primitive is illegal.
Values: cooperation class identifier(s) or ANY. Default: ANY.

subdirectory right restricts how cd-nodes and capabilities contained in the subdirectory may be used; each right may be ON (active) or OFF (inactive); the rights are: TRANSFER, COPY, REGISTER, REMOVE, HOLD, MERGE, VIEW-CAP, VIEW-NODE, MODIFY, DESTROY-MANAGER-NODE, DESTROY-DIR-NODE, CHANGE-DIRECTORY, CREATE-PORT, and CREATE-TYPE.
Value: each is active or inactive. Default: all are active.

subdirectory id a pointer, global name, to the subdirectory cd-node corresponding to this capability.(Cannot be modified through the **Modify** primitive - it is generated by the kernel upon creation of the capability.)

The capcaps of the subdirectory capability are: COPY, TRANSFER, REGISTER, REMOVE, HOLD, DESTROY-NODE, MERGE MODIFY-CAP, MODIFY-CAPCAP, VIEW-NODE and VIEW-CAP.
Value: each is active or inactive. Default: all active.

Figure 8: Contents of a Subdirectory Capability

named in the capability. Since an operation capability can be used to create a port to access a protected object, a manager definition capability signifies the privilege to provide other processes with specific types of access to their objects. This effectively is the privilege to control access to the object's type.

The primary kernel primitive associated with the subdirectory capability (figure 8) is **Change-directory**. The Change-directory primitive is used by a process to change its active directory to the subdirectory named in the subdirectory capability.

The subdirectory capability also contains a set of *subdirectory rights* (figure 9). When a subdirectory capability is exercised to make a subdirectory active, the subdirectory rights override the capcaps of each individual capability, and further restrict the use of the capabilities registered in the subdirectory. This restriction during the changing of

COPY a capability residing in the subdirectory cannot be copied;

TRANSFER a capability residing in the subdirectory cannot be transferred;

MERGE a capability residing in the subdirectory cannot be associated with any other compatible capability in order to augment its capcaps and/or rights.

REGISTER no capabilities may be registered in the subdirectory;

REMOVE no capability residing in the subdirectory may be removed from the directory;

HOLD no capability in the subdirectory may be made transient;

DESTROY-MGR-NODE no manager definition cd-node residing in the subdirectory may be destroyed;

DESTROY-DIR-NODE no subdirectory cd-node residing in the subdirectory may be destroyed;

VIEW-NODE the contents of none of the cd-nodes residing in the subdirectory can be examined;

VIEW-CAP no capability in the subdirectory can be examined;

MODIFY no capability or cd-node in the subdirectory may be modified;

CHANGE-DIR no process may exercise a subdirectory capability residing in the subdirectory to change to a new active subdirectory.

CREATE-PORT no process may exercise an operation capability residing in the subdirectory to create a port to request a service.

CREATE-TYPE no process may exercise a manager definition capability residing in the subdirectory to create new operation capabilities.

Figure 9: List of rights associated to a subdirectory and their meaning in the case they are inactive

the active directory is referred to as *privilege filtering.* Privilege filtering allows a fine granularity of control over the use of capabilities within an active directory. This is vital for supporting an effective mechanism that allows processes to switch from one protection domain to another dynamically. In this situation, when a process wants to switch to a new protection domain, it has to traverse the capability directory and change to a new active directory. While traversing the capability directory, a process may have to visit intermediate subdirectories which contain capabilities that the process need not be authorized to exercise or even view. By deactivating all rights except the CHANGE-DIR along the path between the initial and goal subdirectory, the mechanism for switching to a new domain becomes simple, and the security of the system is not compromised.

From the above, it is evident that a path in the graph structured capability directory, along which a process switches from one protection domain to another, corresponds to a set of subdirectory capabilities. Thus, a stable subdirectory capability links one subdirectory cd-node and another. Although a stable manager definition capability (as well as a stable operational capability) connects a subdirectory cd-node and a manager definition cd-node, they cannot be traversed. These connections are used to 'point to' the manager definition of interest.

It should be noted that rights associated with an active directory do not affect the rights defined in the subdirectory capabilities contained in the active directory. In this sense, rights are not propagated along the path reaching an active directory and that their scope is that subdirectory pointed to by the subdirectory capability in which they are defined.

Another feature in Gutenberg, that is a byproduct of the notion of rights, is the possibility of defining *private subdirectories.* The capabilities contained in a user's private subdirectory may only be exercised by that user's processes [5]. The rights of a private subdirectory can be such that other users can register capabilities in it. This allows a process to give its privileges to another without explicitly passing them through ports.

When a process invokes the change-directory primitive to change its active directory, a copy of the used subdirectory capability, called *active directory capability,* is created and saved in the process' PCB, while the previous active directory capability is destroyed. The capcaps of the active directory capability are modified accordingly to reflect the restrictions derived from the rights in the use and manipulation of the active directory capability and active directory cd-node themselves. Any reference to a capability is now with respect to the active directory capability in the PCB, eliminating the need for traversing the entire path starting from the root subdirectory.

[5]Each user in Gutenberg is associated with a private subdirectory, called *primary* subdirectory. The primary subdirectory of each user is registered in the root subdirectory of the capability directory, the latter being accessible only to the kernel. When a user process is instantiated (at login time), its primary subdirectory becomes its active directory.

Cooperation Class Provides a wild card privilege that may be merged with any other capability type. Thus, the primitive is ANY. It has two attributes:

class name user-specified name used to identify the cooperation class in the current protection domain.
Value: a name. Default: none; must be specified upon creation.

class id system wide unique identification of cooperation class. (Cannot be modified through the **Modify** primitive - it is generated by the kernel upon creation of the capability.)

The capcaps of the cooperation class capability are: COPY, TRANSFER, HOLD, REGISTER, REMOVE, MERGE, VIEW-CAP, MODIFY-CAP, and MODIFY-CAPCAP Value: each is active or inactive. Default: all active.

Figure 10: Contents of Cooperation Class Capability

As it was previously stated, both subdirectory and manager definitions cd-nodes are identified by user-specified names. These user-specified names need to be unique in the active directory of a process. These names are stored in the corresponding subdirectory and manager definition capabilities and can also be used by processes to identify the capabilities themselves. In the case of the other two types of capabilities, a value of other attribute serves to identify the capability; for example, the name of the operation in an operational capability. Processes may also identify a capability by describing it. This is particularly useful in identifying derivable capabilities from others in a protection domain, without explicitly creating them. In addition, from a programmer's point of view, it frees the process from remembering capabilities' user-specified names in different contexts, although this would make the search faster.

The fourth type of capability viz the cooperation class capability, abstractly, represents the privilege of a process to participate in a cooperative activity identified by a unique identifier, the *class id* (figure 10). The cooperation class capability may be associated with any other capability type, thus providing a wild card privilege. Hence, the primitive associated with this capability is ANY denoting its wild card property. When a cooperation class capability is associated with either a subdirectory or a manager definition capability, it restricts the invocation of the corresponding primary primitive to the processes which possess the cooperation class capability. In effect, such processes become members of the cooperative activity represented by the capability. In this way cooperation class capabilities complement the role of capcaps by determining how these capabilities may be exercised. In the case in which a cooperation class capability is associated with an operation capability, it specifies a cooperative activity, for example a communication with a manager of a shared object. Here, the cooperation class capability can be used as a synchronization token or to identify either an instance of a manager or a particular instance of an object. It could also be used to identify a transaction, a file, or a process. It can be used to classify the users in the system into groups and divisions for administrative reasons. In these cases, the mapping of the class id to the object is done by the manager, not by the kernel. New cooperation class capabilities can be created, on request, by the kernel.

In summary, four types of stable capabilities can be stored in the graph structured capability directory. The selection of a graph structure over a simpler one, i.e. tree structure, for the capability directory is justified by the de facto sharing of subdirectories, that can be controlled through the rights attached to the paths for reaching a subdirectory. This would not logically follow in the case of a tree structure in which only one path exists for reaching a subdirectory.

3.4 Transient Capabilities

Port capabilities and copies of stable capabilities from the capability directory are the transient capabilities. Two features distinguish the transient capabilities from the stable capabilities: they are owned by a single process, therefore, they cannot be shared, and their existence is dependent on the existence of the process that owns them. All the transient capabilities a process owns exist only for the duration of the process existence and are destroyed when the process terminates. The transient capabilities possessed by a process, are stored in the process' *capability list*, or *c-list*. Capabilities in the c-list, as previously stated, together with the active directory of a process, form the protection domain of the process. Thus, a process in addition to the capabilities in its active directory, may exercise the capabilities in its c-list. In particular, any search for a capability starts from a process c-list and if it fails, proceeds to the process active directory. There is a way in which this order of the search can be reversed. This is desirable in the case that a process wants to exercise a capability in its active directory, but a different capability with the same name exists in its c-list. This definition of the order of the search can be avoided in some cases, if the process identifies the capability by description. Clearly, a process can also resolve this ambiguity of names between its c-list and its active directory by changing the user-specified local names in its c-list, an operation that a process is always allowed to perform. Capabilities in a c-list have unique names. This is also the case for capabilities in a particular subdirectory. The above makes it apparent that a capability with the same name may exist both in the active directory and the c-list of a process.

A transient capability comes into existence whenever a process holds a capability in its active directory, i.e. moves it or copies it into its c-list, whenever it receives the capability from another process via a port, whenever it creates a capability by invoking the proper create primitive, or whenever it creates a port. In the last instance, two port capabilities are generated to access the created port, one for the client of the port and the other for the server of the port. The one for the client is placed in its c-list. That of the server is placed in server's c-list, if the server is active, otherwise it is held by the system until the server is instantiated. A process can move a capability in its active directory into its c-list in order not to lose the capability when it changes its active directory to another subdirectory.

The part of the c-list in which the kernel maintains the port capabilities of a process is called the *p-list*. The port capabilities are *exclusive* and as such, port capabilities are inherently transient, cannot be shared or copied. However, either the client or the server process may transfer its port capability to another process. The transferring may be either temporary (the semantics of a lend with the ownership retained) with the SEND-RECEIVE primitive or permanent with the SEND primitive. In the latter, the ownership of the port is transferred along with the port capability (see section 4.3).

The only capcap associated with a port capability is *transfer* which is set by the kernel upon creation of the port. The transfer capcap in the client's port capability is set to the value of the corresponding transfer capcap in the operation capability used. If the used operation capability is stable, then the transfer right of the client's active directory is also taken into consideration. The transfer capcap in the server's port capability is always set. A server process may transfer a port that it serves, to another server as long as the port functionality is preserved. This is essential for supporting the implementation of load balancing algorithms and realizing an hierarchy of object managers in both system and user's level, features which improve the performance and reliability in a distributed system. A common example of a hierarchy of managers is a manager structured based on the master/slave model, in which a process can only create a port for requesting an operation to the manager's master process. The master process decides which slave process should service the request and passes the port capability to it. A process may reset the transfer capcap of the port capability when it passes the capability to another process, to prevent further transferring.

Transient capabilities contribute to the flexible use of capabilities in Gutenberg without compromising the security of the system because the c-list is saved in the PCB and may only be manipulated through kernel primitives.

The kernel primitives that manipulate the capabilities both within a c-list and an active directory, fall into two classes: *generic* and *special primitives*. Generic primitives are further classified into *constructive* in that they do not affect the participating capabilities, and into *destructive*. Constructive primitives are designated by the ending -C attached to their names. Recall that a number of capcaps and rights correspond to each primitive, and must be active in the capability on which the primitive is invoked, and the active directory of the invoking process. The eleven function types are: *Create*, *Register*, *Register-C*, *Remove*, *Hold*, *Hold-C*, *Drop*, *View*, *Modify*, *Merge* and *Merge-C*. Here is a brief description of these generic kernel primitives:

Create These primitives creates a capability. **Create-port**, and **Create-operation** are instances of this generic primitive. A **Create** always creates a transient capability with full privileges, which may then be registered or transferred with all or part of these privileges retained. When **Create** is used to create a capability the parameters that have to specified for the create correspond to the attributes of the capability that do not have a default value. Values of parameters may be changed with the **Modify** primitives. A process issuing the **Create-operation** primitive must possess the proper manager definition capability. If the manager definition capability is stable, in addition, the process must have the **Create-type** right for its active directory. Creating a port capability requires an operation capability. If the operation capability is stable, the process must have the **Create-port** right for its active directory. Other **Create** primitives require no special privilege.

Register These primitives make a transient capability, or one derived from transient capabilities, stable. A process may reset part of the capcaps and/or rights of a capability, when it registers the capability. These primitives are destructive in that they remove the transient capabilities involved. The purpose of these primitives is two-fold: to allow a process to store a capability for future reference in another session; and, to allow a process to share a capability with other processes which are not currently instantiated, but share access to a subdirectory. They require that the process have the **Register** right for its active directory, and that the transient capability or the transient ones involved in the derivation, have the REGISTER capcap active. In addition, the kernel enforces the restriction that a capability cannot be registered, if its TRANSFER capcap is active while its COPY capcap is inactive. This is necessary for supporting effective privilege transfer mechanisms discussed in the next section.

Register-C These primitives are similar to the **Register** ones but for the following differences: They are constructive in that they create a new capability and don't affect the transient capabilities involved. They require that the process have the **Register** right for its active directory, and that the transient capability, or the transient ones involved in the derivation, have both the REGISTER and COPY capcaps active.

Remove These primitives delete a stable capability from the active directory. They require that the process have the **Remove** right for its active directory, and that the stable capability have the REMOVE capcap active.

Hold These primitives make a stable capability, or one derived from stable capabilities, transient. A process may reset part of the capcaps and/or rights of a capability, when it holds the capability. Hold primitives are destructive in that they remove the stable capabilities involved from the process active directory. The purpose of these primitives is to allow a process to retain a capability for its active directory when it changes its active directory to another subdirectory. They require that the process have the **Hold** right for its active directory, and that the stable capability or the stable ones involved in the derivation, have the HOLD capcap active. If the derived one to be held is an operation capability, they require, in addition, that the process have the **Create-port** right, for its active directory.

Hold-C These primitives are similar to the **Hold** ones but for the following differences: Hold-C are constructive in that they create a new capability without affecting the stable capabilities involved. They require that the process have both the **Hold** and **Copy** rights for its active directory, and that the stable capability or the stable ones involved in the derivation have both the HOLD and COPY capcaps active.

View These primitives bring a copy of a transient or stable capability, or a cd-node into the address space of a process. Partial views are also facilitated in the case of a subdirectory; it is possible to bring capabilities of a specified type into the address space of the process. The purpose of these primitives is to allow a process to examine capabilities it possesses, and, if desired, use this information to modify or create new capabilities. If a capability is being viewed, VIEW-CAP must be one of the active capcaps of the capability. If a cd-node is being viewed, the VIEW-NODE capcap must be active in the capability pointing to the cd-node. If the capability is stable, in addition, the process must have the **View-Cap** and the **View-Node** rights for its active directory.

Modify These primitives allow a process to modify an existing transient or stable capability, or a cd-node. If a stable capability or a cd-node pointed by a stable capability is being modified, the requesting process must have the **Modify** right for its active directory. If non-capcap attributes of a capability is being modified, MODIFY-CAP must be active in the capcaps of the capability. For the capability's capcaps to be modified, MODIFY-CAPCAP must be active in the capcaps of the capability. If a cd-node is being modified, MODIFY-NODE must be active in the capcaps of the capability pointing to the cd-node. A process may modify the local name of a transient capability as long as it is unique within the c-list.

Merge These primitives allow a process to merge two compatible capabilities to obtain another. Two capabilities are compatible if they are of the same type and have identical non-modifiable attributes (attributes whose values cannot be changed with the **Modify** primitive). The resulting capability is also compatible with the participating ones. The cooperation class(es) attributes are sets of cooperation capabilities with ANY representing all the allowable capabilities. The cooperation class(es) attribute of the resulting capability is obtained from the intersection of the corresponding attributes of the two participating capabilities, while the capcaps and the rights are obtained from the union of the capcaps and rights of the capabilities involved. Merge primitives are destructive in that they replace one of the participating capabilities with the obtained one. The purpose of this primitive is to allow a process to combine two capabilities into one with augmented capcaps and/or rights. They require that the process have the **Merge** right in its active directory if stable capabilities are involved, and that the participating capabilities have the MERGE capcap active. If both the participating capabilities are stable (transient), then the resulting one is also stable (transient). If one of the participating capabilities is stable, while the other is transient, the process specifies the kind of the resulting capability. In the case that the resulting capability is specified to be transient, these primitives require, in addition, that the process have the **Hold** right for its active directory and that the stable capability have the HOLD capcap active. If, on the other hand, the resulting capability is specified to be stable, these primitives require, in addition, that the process have the **Register** right for its active directory and the transient capability have the REGISTER capcap active.

Merge-C These primitives are similar to **Merge** ones but for the following differences: Merge-C primitives are constructive in that they create a new capability without affecting the participating ones. They require that the process have both the **Merge** and **Copy** rights in its active directory if stable capabilities are involved, and that the participating capabilities have both the MERGE and COPY capcaps active.

As has previously been discussed, the manager definition and subdirectory capabilities are slightly different than other capabilities. These capabilities contain pointers to manager definition and subdirectory cd-nodes, respectively. When one of these capabilities is created with the **Create** primitive, the cd-node is created as well. No special privilege is required to create a cd-node, though to store the corresponding capability in a subdirectory requires privilege. These cd-nodes exist in the system until either they are explicitly destroyed by using the proper **Destroy** primitives, or all of the capabilities that point to them are deleted using the **Remove** or **Drop** primitives.

The kernel uses the c-list and the active directory of a process to check whether it has a legitimate privilege for executing a primitive whenever a process requests its execution. Therefore, the consistency and availability of the capability directory is fundamental to the correct operation of the protection mechanism. As is commonly done with resources in distributed systems, the capability directory is physically distributed, though still logically unified, across the distributed system; for a discussion of issues involved in the distribution and the approach adopted see [**Ramamritham et al. 85**].

4 USER-DEFINED OBJECTS

Recall that the Gutenberg kernel is not involved in the protection of objects that are purely local to a process. User-defined objects are those objects managed by one process but accessible by other processes via operations requested using ports. Here, we discuss how user-defined objects are created, shared and protected.

4.1 Type Creation

The representation of a user-defined type in Gutenberg is the manager definition cd-node (figure 4). A manager definition is created by a process invoking the **Create-manager** primitive. Recall that no special privilege is required to invoke this primitive. In invoking the primitive, a process must provide five parameters: A cooperation class capability identifying the file containing the executable image for the process; a subdirectory capability identifying the *default directory,* the subdirectory which becomes the active directory of any process instantiated from this manager definition; the *operation list,* the specification of the operations that are implemented by the manager; the *manager initiation protocol,* indicating how manager processes are instantiated; and, the *manager dependency* indicating whether a manager process will be destroyed when all ports connected to it are destroyed.

Upon successful execution of the **Create-manager** primitive, the kernel places a manager definition capability, pointing at the new manager definition and containing its name, in the process's c-list. After the manager definition and the corresponding manager definition capability are created, the process may use the manager capability to invoke the **Create-operation** primitive to create the operation capabilities for the type. These operation capabilities can then be stored in the capability directory or distributed over ports to processes wishing to use the type.

4.2 Manager Initiation Protocols

The manager initiation protocol specified when creating a manager definition determines the manner in which ports are connected to manager processes. It specifies whether all the object instances of a type are managed by one process or each object is managed by different processes. There are three initiation protocols in Gutenberg: *conservative*, *creative* and *class conservative.* In the conservative manager initiation protocol, a manager process is instantiated from the manager definition only if there is no other manager process executing in the system which was instantiated using this manager definition. If such a process already exists, the port being created is attached to this process. This protocol provides the means to produce a manager process that manages all the objects of a type, and to automatically connect port-creating processes to this manager. Using this protocol, the manager can be informed of the object being accessed at port-creation time using a cooperation class capability. Instantiating more than one conservative manager requires creating more than one manager definition.

The creative protocol creates a new manager process from the creative manager definition cd-node for each new port created. This protocol allows a process to create a port to a new process under all situations. This protocol allows a process to isolate the newly created manager in order to ensure that the manager cannot leak information. However, it cannot support multi-port interconnections between a specific client and a specific server.

The third protocol is the class conservative manager initiation protocol. It allows new managers to be instantiated selectively based on the cooperation class capability supplied at port creation time. The class conservative manager is typically designed to manage one object of the type, and may serve multiple ports from any number of processes.

In the class conservative protocol, when a port is created using an operation capability, the kernel checks to see if a process associated with the specified class id has been instantiated from the manager definition pointed to by the operation capability. If so, the port is connected to this manager. If not, a new manager is instantiated and associated with the specified class id. Note that class id is used by the kernel in the selection of a manager process, not to identify an object. Upon instantiation of a new manager, the kernel places a copy of the cooperation class capability used in the manager's c-list, the manager process being part of the cooperative activity identified by the class-id.

Both conservative protocols allow any two processes to communicate indirectly through a conservative process, although they cannot establish direct, full duplex interconnections. However, using these protocols and by allowing processes to pass port use privileges via ports, the one-to-one process intercommunication topology adopted by Gutenberg can be expanded to arbitrary topologies (see next section for an example). For a more detailed description of manager initiation protocols, see [**Stemple et al. 86**].

4.3 Object Protection and Sharing

Once a type is created, distributing privileges to allow other processes to use the object type corresponds, in Gutenberg, to distributing operation capabilities linked to the manager definition. As has previously been discussed, for a process to access a shared resource, a port is needed between itself and the process managing the object. Establishing a port involves checking for an operation capability in the active directory or c-list of the requesting process. Thereafter the kernel performs access authorization for a user-defined operation simply by checking that the requesting process has the privilege to access the port associated with the operation. Thus, creating and accessing user-defined objects involves using kernel-defined capabilities to authorize access to kernel-defined objects, and does not involve checking user-defined capabilities as in other capability-based systems.

This, in conjunction with the adoption of a non-uniform model, contributes to the efficacy of the Gutenberg approach to protection.

In Gutenberg, the ability of a process to access shared objects can change dynamically through the transfer of capabilities on ports. There are three ways in which one process can transfer some of its capabilities to another; in all cases capabilities are transferred over a port connecting the sending and receiving processes. The transferred capabilities are always placed on the receiving's process c-list. Since the kernel is the manager of the capabilities and ports, it monitors the transfer of capabilities between processes.

The first method of capability transfer is the transfer of a port capability. The sending process loses the port capability, and therefore the privilege to execute the object operation associated with the transferred port; the receiving process obtains this privilege.

The second method of capability transfer is the transfer of an operation capability (which can be associated with a cooperation class) that can be used to create a port to access an object (identified by the cooperation class). In fact, the receiving process may use the operation capability to create many ports.

The third method of capability transfer is by registering the capability to be transferred in a subdirectory and transferring the subdirectory capability that points to it. The receiving process can use the subdirectory capability to make the subdirectory its active directory, thereby allowing it to use the capabilities in the subdirectory. Using this method a process may transfer a number of capabilities at once with the minimum communication overhead.

In all three methods, capabilities are transferred either by SEND as part of the message or by SEND-RECEIVE as part of the request details or the return message. Upon invocation of either of these two port primitives, the kernel validates the capability transfer. A process may specify the capabilities to be transferred, either by name or by a description. The latter is equivalent to building the capability in its address space. If the specified capability already exists in either the sender's c-list or active directory, then it is transferred. Otherwise, the kernel attempts to derive it from the existing capabilities. In any case, for a capability to be transferred, it is required that the TRANSFER capcap of the capability or the capabilities involved in the derivation of the capability, to be active. If stable capabilities are involved in the transferring, it is required, in addition, that the sender has the **Transfer** right for its active directory. Whether the sender loses the transferred capability is determined by the kind of the capability, and the value of its COPY capcap. Specifically, exclusive capabilities, i.e., port capabilities and transient capabilities with their COPY capcap inactive, are removed from the c-list of the sender and placed in the receiver's c-list. Non-exclusive capabilities, stable capabilities and transient capabilities with their COPY capcap active, are never removed from the possession of the sender. That is, a copy of the capability to be transferred is placed in the receiver's c-list.

As elucidated in the previous section, the two mechanisms of capability transfer are distinguished by the semantics of the transferring. The transfer by SEND is permanent and the transferred capability is not returned to the sender, whereas transfer by SEND-RECEIVE is temporary and the transferred capabilities can be held only while the recipient is processing the request to which it pertains. When the recipient executes the SEND that satisfies the request, the kernel automatically returns the *outstanding* capabilities to the process which executed the SEND-RECEIVE. If the capability to be returned at the time of SEND is not possessed by the recipient of the SEND-RECEIVE, the SEND is rejected. Since the recipient of a capability via a SEND-RECEIVE may transfer the capability to another process, if the transfer capcap is active, to avoid the anomalous situation wherein the recipient of a capability is permanently unable to possess the capability, exclusive capabilities as well as ones with the COPY capcap inactive, received in request details, are restricted to be further transferred only as part of request details of a SEND-RECEIVE, and never as part of a message sent by a SEND. The kernel in addition to this rule, enforces two more restrictions on the SEND primitive. The first is that the client privilege cannot be separated from the destroy privilege when it is transferred using the SEND primitive. On the contrary, if the transfer is done by means of SEND-RECEIVE, the destroy privilege can never be passed since the exercise of it by the recipient would make the returning of the capability meaningless. Moreover, the owner of a port may destroy the port provided it possesses the port capability at that time. This restriction prevents the owner from sabotaging another process's use of the port. The second restriction is that if a capability is passed using SEND, then it must be an acknowledge-SEND. This is essential for furnishing recovery from a failed transmission before the receiver has removed the message. Figure 11 summarizes the rules and restrictions on capability transfer in Gutenberg.

Exercising the **Port-destroy** primitive has ramifications in the case of a Send-receive port, on which there is a pending request and outstanding capabilities, some of which themselves may have pending requests and possibly outstanding privileges. In order to avoid any inconsistencies, the system waits for all pending requests to be cleared on the ports that are outstanding, and when the server of the port to be destroyed performs the SEND, the kernel returns the outstanding capabilities, ignores the sent message and destroys the port.

The two issues of concern when transferring capabilities are the ***sharability*** and ***stability*** of the transferred capabilities. Obviously a capability is sharable by the receiving process if the receiver can further transfer it to another process. Through the use of *capcaps*, the sending process has control over whether a capability can be registered or further transferred. In general, a process is unrestricted to reset part or all of the capcaps and/or rights of a capability, when it transfers the capability. This does not imply that a process loses any of its privilege derived from the capcaps and rights, if the process passes the capability once with these capcaps and rights inactive with a SEND-RECEIVE primitive.

General

For a process to transfer a capability, the capability must contain the transfer capcap.

A port on which a capability is to be transferred must be typed to allow the transfer; type information is contained in the operation capability used to create the port.

If port capability is to be transferred, the transferring process must have no pending request on the port.

Transferred capability cannot be revoked after it is received.

Transfer of capability using SEND

SEND must be acknowledge-SEND.

If capability is exclusive, sender loses capability permanently.

If client capability of a port is transferred, the owner (destroy) privilege must also be sent; the receiver becomes the owner of the port.

If an exclusive capability is being transferred, it cannot have been previously received as a part of request details i.e., by virtue of a SEND-RECEIVE.

Transfer of capability using SEND-RECEIVE

If the transferred capability is exclusive, sender (executer of SEND-RECEIVE) loses the capability temporarily; kernel assures return of capability.

Destroy privilege to a port may not be transferred.

Figure 11: Summary of the rules and restrictions on capability transfer

Finally, we show, here, with an example the necessity of the additional restriction on the Register primitives. The kernel does not allow registering a capability with the TRANSFER capcap active and the COPY capcap inactive. Suppose that this was allowed, and such a capability was registered in a subdirectory which is the active directory of a process. At some point, the process transfers this capability as request-details of an asynchronous SEND-RECEIVE, and immediately afterwards changes its active directory by invoking the **Change-directory**. When the request is satisfied by the recipient executing the SEND, the passed capability is returned automatically to the sender and placed in its active directory which is different from the one at the time of the transferring of the capability. This clearly is an erroneous situation which should be prevented. By imposing the restriction discussed earlier this is achieved in a simple way without the need for back-pointers or any other extra information. For the same reason, the **Transfer** right of the active capability is reset in the case that the **Copy** right is inactive at the creation time of the capability during the execution of the **Change-directory** primitive. For more detailed discussion of the privilege transferring mechanisms as well as their implications see [**Ramamritham et al. 86**].

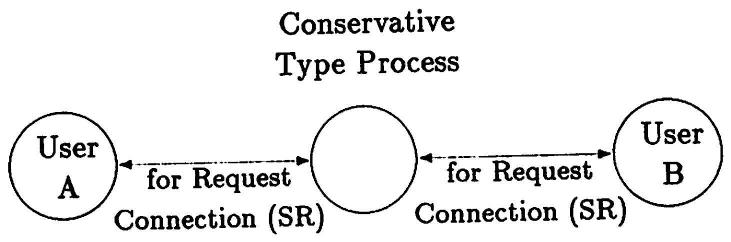
5 AN EXAMPLE: INFORMATION-EXCHANGE FACILITY

We conclude with an example of how a information-exchange facility which provides full duplex communication between two arbitrary processes as well as a conference among many processes, can be implemented using the Gutenberg mechanisms. This facility is particularly useful in many environments such as automated office.

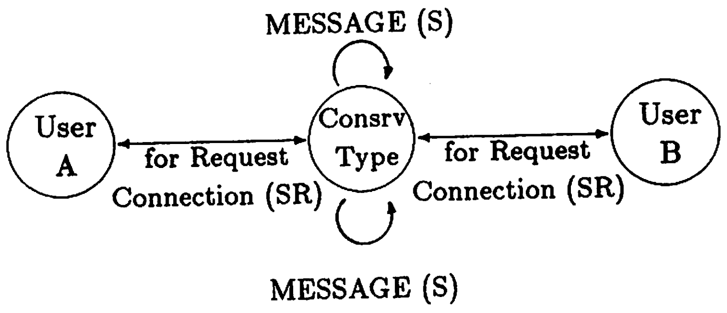
In order for two arbitrary processes to exchange messages, it is required that two ports be established connecting these processes. The connection is made in such a way that one process may send a message to the other one using one of the ports and receive a message sent from the other process through the second port. In the Gutenberg system such a direct port creation between two independent processes is not possible, since all ports are created based on the functional addressing principle, and not by identifying a particular process. (This would only be possible, if the communicating processes were both of a conservative type and each of them possessed the appropriate capability to create a port to the other one.)

Two arbitrary processes, however, can indirectly communicate through a conservative or class-conservative process to which they are both connected. It follows that such a conservative type process which serves both of the independent processes, can be used to create two ports to itself and afterwards pass both the client and server ends of ports to the two processes, making them complementary clients and servers of the ports. The two independent processes that wish to communicate are required to establish a Send-receive port to the conservative process for the purpose of receiving the two port capabilities (figure 12). The ownership of the ports are passed along with the client capability. The name of

(a)



(b)



(c)

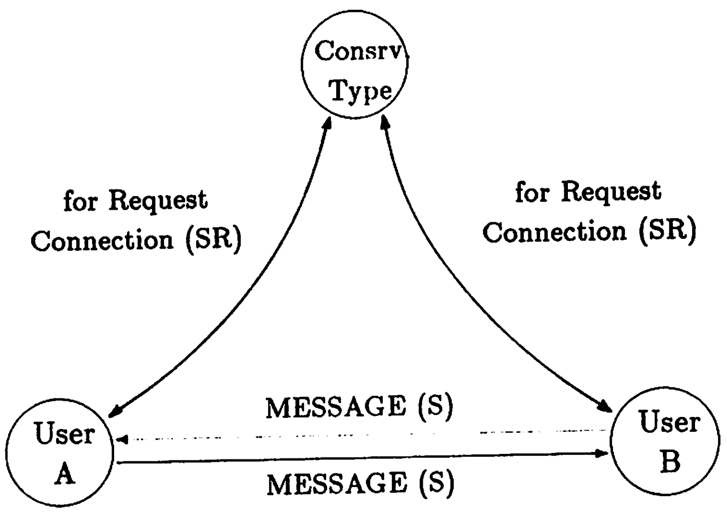


Figure 12: Stages in establishing full duplex connection
(a) before any request for connection. (b) after both users requested a connection passing each others names. (c) after the concervative type process passed the two ports typed MESSAGE to the user processes, making them complementary clients and servers of the ports.

the process for which the connection is requested, is passed as part of the request-details. These names which processes use to identify themselves, are chosen by the human users and have no relationship to the system process identifiers.

An issue of concern, when forming the connection, is the validation of the user-defined process names and of the connection. Connections should be established, if the communication between the two parties is allowed.

Now, we present the complete information-exchange facility which uses the system identification of users based on their password-protected primary (login) subdirectories to validate process names. Participation in a particular communication group or conference is identified by a specific cooperation class id.

The information-exchange system consists of a conservative process, called *main switchboard*, and a number of class-concervative processes, called *group switchboards* and *conference managers*. It is basically a hierarchy of managers analogous to the telephone system whose local exchange and broadcast exchange (PBX or ABX) correspond to our group switchboard and conference manager, and main switchboard respectively. The group switchboards are responsible for establishing the duplex connections between two independent processes whereas the conference managers are responsible for establishing the conferences between many users. All group switchboards and conference managers are connected to the main switchboard with a Send port created by the main switchboard. In fact, a group switchboard or a conference manager is instantiated as a result of the creation of one of these connecting ports with the main switchboard. Figure 13 shows the active directories that the main switchboard and every group switchboard and conference manager must have in order to establish the interconnections.

Each user identifies himself or herself to the system by supplying a username and a password. Upon successful login, a user process is instantiated for the user with the user's primary subdirectory being the active directory. During login time, the login process creates two Send-receive ports, one for requesting a *connection* (in the figures is referred as CONNEC) and the other for requesting a *conference* (in the figures is referred as CONFER), to the main switchboard and attaches them to the newly instantiated user process. At the same time, login process informs the main switchboard about the attached ports and username association by executing a SEND over a port connecting them.

A group of users who are allowed to communicate, are identified by cooperation class ids. Each user possesses the cooperation class capabilities that represent the communication groups in which the user is a member. These capabilities get into the user's possession either by receiving them over ports or by holding them from subdirectories accessible to the user. These may be further registered in the user's primary subdirectory or in other subdirectories accessible from its primary subdirectory.

Two users must belong to one common group in order to communicate. Direct, full

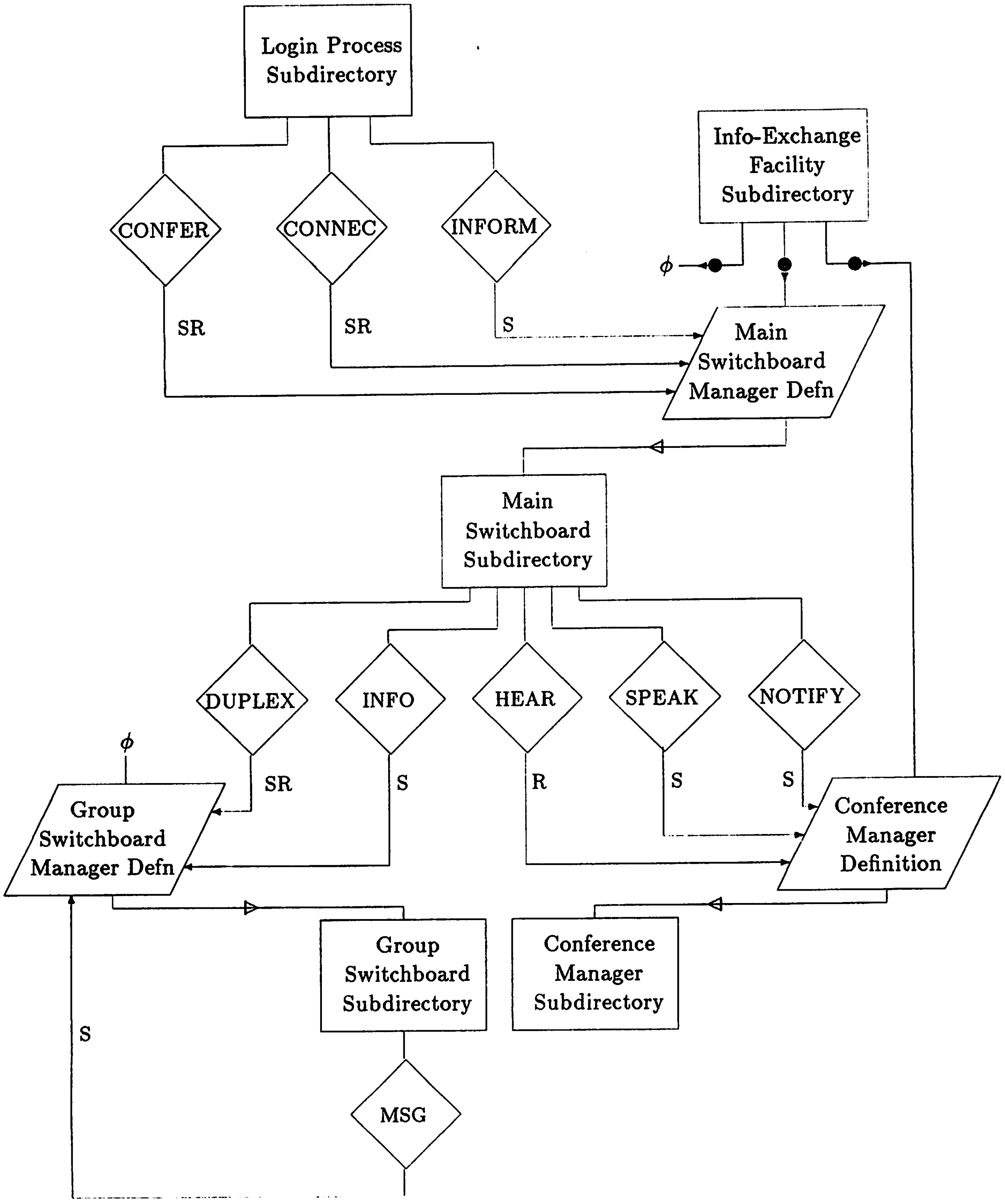

Login Process Subdirectory
CONFER
CONNEC
INFORM
SR
SR
S
Info-Exchange Facility Subdirectory
ϕ
Main Switchboard Manager Defn
Main Switchboard Subdirectory
DUPLEX
INFO
HEAR
SPEAK
NOTIFY
ϕ
SR
S
R
S
S
Group Switchboard Manager Defn
Conference Manager Definition
Group Switchboard Subdirectory
Conference Manager Subdirectory
S
MSG

Figure 13: Capability Directory Configuration needed by the Information Exchange Facility

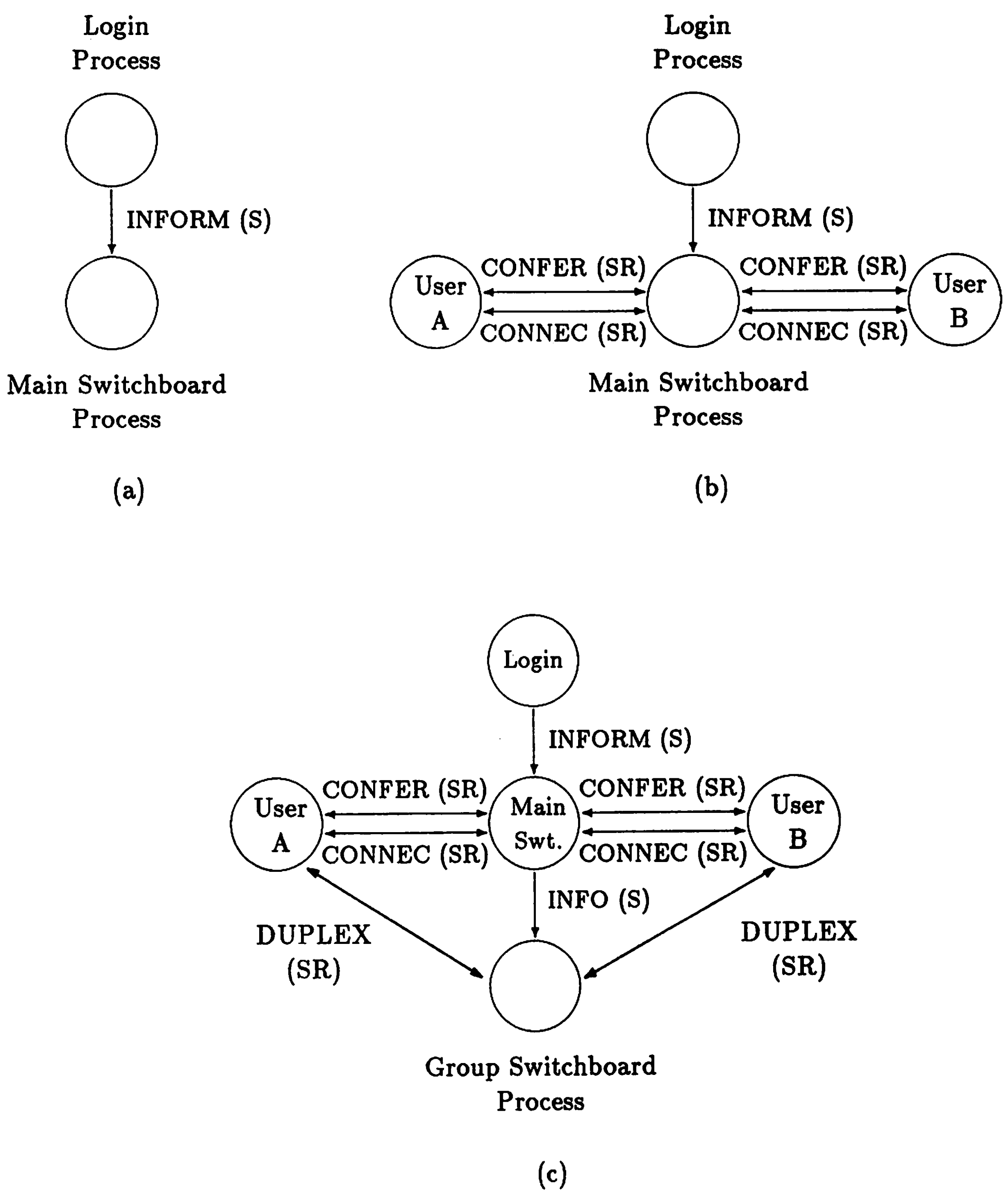


Figure 14: First three Steps in establishing a two-way communication in the Information Exchange Facility:
(a) before any user login in the system. (b) after two users login in the system (c) after both users executed a SEND-RECEIVE over the port typed connection, pasing the same cooperation class capability.

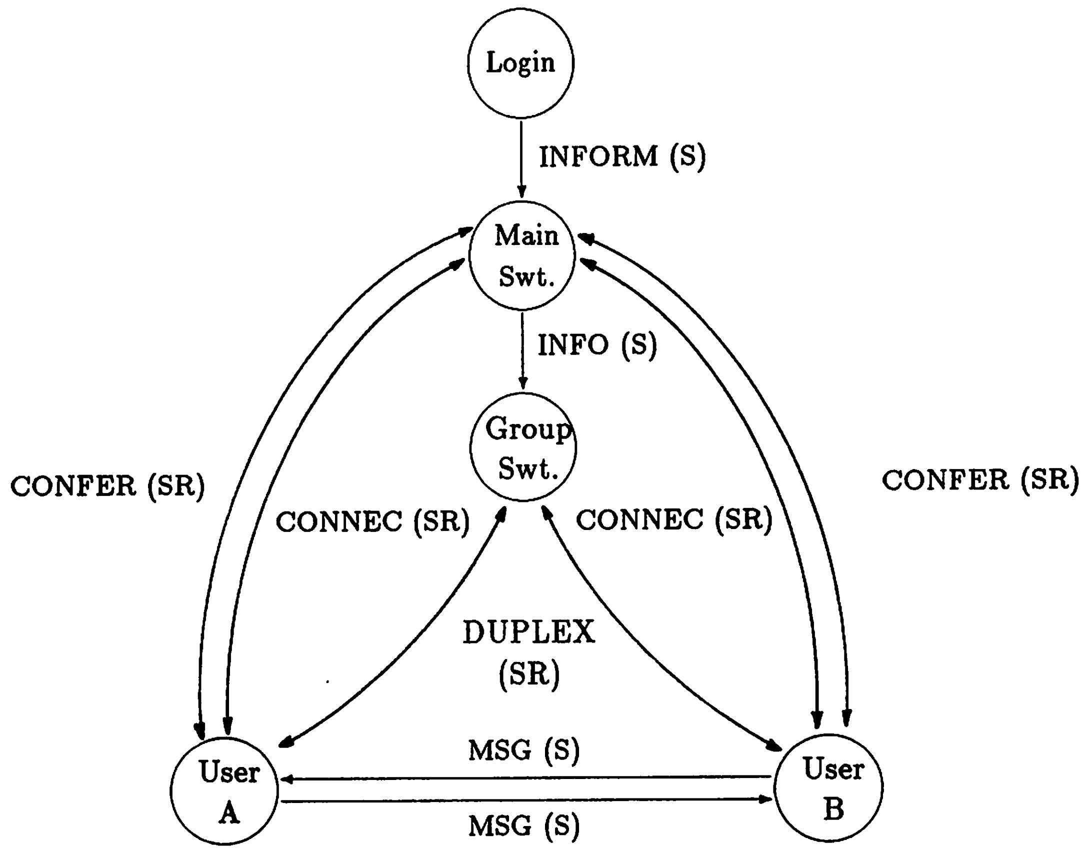


Figure 15: A typical process configuration in a two-way communication provided by the Information Exchange Facility.

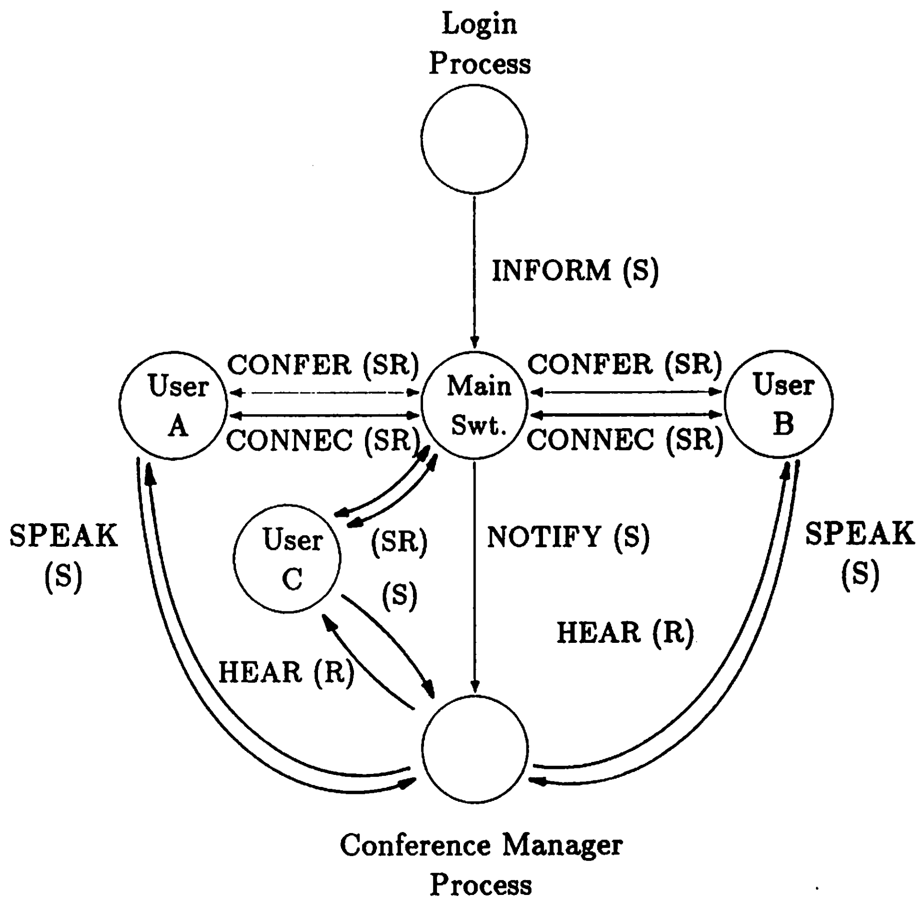


Figure 16: A typical process configuration in a Conference provided by the Information Exchange Facility.

duplex connections between users belonging to the same group can only be established by the group switchboard in the way discussed in the beginning of this section. First, a user process can be connected to a group switchboard through the main switchboard. The user requests the main switchboard to establish a connection to a group switchboard by executing a SEND-RECEIVE on the port typed connection, and passing a cooperation class capability as part of the request-details. The main switchboard uses the cooperation class to create a Send-receive port for requesting *duplex-connection* (in the figures is referred as DUPLEX) to the group switchboard that is identified by the cooperation class. It satisfies the request by SENDing the user's end of the created port to the requester (the passed cooperation capability is automatically returned by the kernel upon execution of the SEND) and by informing the group switchboard about the port and username association. Figures 14 and 15 illustrate the steps required to establish a two-way communication between two processes.

Conference among many users are realized by the information-exchange facility in a similar way as in the direct full-duplex communication. Each conference is identified by a cooperation class. During login time, in addition to the Send-receive port for requesting a connection, the user process is connected to the main switchboard with another Send-receive for requesting a conference. When a user wishes to participate in a conference in which it has the privilege to do so, it executes a SEND-RECEIVE operation on the port typed conference, passing a cooperation class capability. The cooperation class capability specifies the conference of interest. As in the case of the request for a connection, the main switchboard uses the cooperation class capability to create two ports, one Send and one Receive, to the class conservative process that manages the conference. Subsequently, the owner's end of both the created ports are sent to the process which has executed the SEND-RECEIVE operation. The main switchboard again informs the conference manager about the Send port and the username association. There is no need to associate the Receive port with the username since it is just used as an input device. The Send port that connects the main switchboard and the conference manager is created upon the initiation of the conference. When the first user wishing to participate to a conference passes the cooperation class capability, the main switchboard uses it to create this Send port which causes the instantiation of the manager process. In figure 16 we show the configuration of ports and processes after three independent processes have established a conference.

A conference manager is merely a message re-transmitter. When a participant in the conference sends a message, the conference manager re-transmits it together with the sender's name, by sending it to all the other participants over the Receive ports. The use of a conference re-transmitter instead of fully connected processes to implement a conference between many processes was based on the following reasons: First, for a process to participate in the conference, it is required that it establish only two ports to the conference manager in our system, whereas in the case of the fully connected processes, two ports are required for every pair of partricipants; this represents a significant overhead. Second, our scheme allow new processes to participate in a conference and for participating processes to withdraw without interrupting other participants in the conference. The former is achieved by establishing the two ports to the conference manager; the latter by destroying the connecting ports to the manager.

The only drawback of our scheme is that it depends on the conference manager with all the consequences that any centralized scheme has. Clearly, this can be solved using any of the known approaches, i.e., having a a second conference manager standing by.

It should be clear that any arbitrary topology can be attained using port passing and conservative type interconnection service processes. However, the major appeal of the scheme presented here is that no special features are required from the kernel and that it can be easily distributed. The distributed implementation follows easily from the analogy with the telephone system. Many such services with different features can exist in the system without requiring specific support in the operating system kernel.

6 CONCLUSION

The Gutenberg system is a novel attempt to facilitate the design and structuring of distributed computations in an understandable, correct and reliable manner. The heart of the system is an efficient protection mechanism at the kernel level in a loosely-coupled system of cooperating nodes. The crux of the Gutenberg approach is the use of port-based communication and protection, non-uniform object-orientation and decentralized access authorization using ports and the capability directory.

In this paper we discussed the design of the Gutenberg kernel. In particular, we presented the kernel primitives, i.e., kernel-implemented operations for manipulating the capability directory and ports. Since process creation and destruction are byproducts of port operations, there are no explicit primitives to deal with processes in Gutenberg. Gutenberg does allow users to construct managers for user defined objects. Such manager definitions are registered in the capability directory.

A Privilege in Gutenberg is represented by capabilities which can have two levels of persistence, transient for those capabilities which persist only as long as an owning process exists, and stable for those capabilities in the capability directory, whose existence is independent of processes. Gutenberg has mechanisms for achieving transfer of privileges represented by both transient and stable capabilities. Some of the highlights of these mechanisms include:

- Using both unidirectional and bidirectional communication primitives and associating them with permanent (unidirectional) and temporary (bidirectional) granting of privileges in order to provide flexibility in privilege granting while keeping the kernel simple.

- Typing ports with respect to the ability to transfer privileges on them in order to expedite communication in cases where no privilege transfer can be made.
- Restricting ports to connecting one client process with one server process in order to simplify interprocess communication in general and the transfer of privileges in particular.

Design and implementation of a *pseudo* Gutenberg kernel (built on top of UNIX) is currently nearing completion. Experimentation with this kernel should provide qualitative evaluation of the advantages of the Gutenberg approach. We are also studying its extension to a distributed kernel with an emphasis on protection issues [**Vinter 85**].

7 REFERENCES

[**Almes 83**] Almes, G. T., 'Integration And Distribution In The Eden System,' Dept. Of Computer Science Technical Report 83-01-02, University Of Washington, January, 1983.

[**Baskett et al. 77**] Baskett, F., Howard, J., Montague, J., 'Task Communication in DEMOS,' *Proceedings of the 6th ACM Symposium on Operating System Principles*, November, 1977.

[**Birrell et al. 82**] Birrell, A., Levin, R., Needham, R., Schroeder, M., 'Grapevine: An Exercise in Distributed Computing,' *Communications of the ACM*, vol 25, no. 4, April, 1982.

[**Cohen et al. 75**] Cohen, E., Jefferson, D., 'Protection in the HYDRA Operating System,' *Proceedings of the 5th ACM Symposium on Operating System Principles*, 1975.

[**Cox et al. 81**] Cox, G., Corwin, W., Lai, K., Pollack, F., 'A Unified Model and Implementation for Interprocess Communication in a Multiprocessor Environment,' Intel Corporation, 1981.

[**Daniels et al. 82**] Daniels, D., Selinger, P., Haas, L., Lindsay, B., Mohan, C., Walker, A., Wilms, P., 'An Introduction to Distributed Query Compilation in R*,' IBM Research Report RJ3497, June, 1982.

[**Dennis et al. 66**] Dennis, J. and Van Horn, E., 'Programming Semantics for Multiprogrammed Computations,' *Communications of the ACM*, vol. 9, no. 3, March, 1966.

[**DeRemer 76**] 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.

[**Dion 80**] Dion, J., 'The Cambridge File Server,' *ACM Operating Systems Review*, October, 1980.

[**Gehringer et al. 79**] Gehringer, E., 'Functionability and Performance in Capability-Based Operating Systems,' Department of Computer Science Technical Report, Purdue University, 1979.

[**Kernighan et al. 78**] Kernighan, B.W., Ritchie, D.M., *The C Programming Language*, Princeton-Hall Inc., Englewood Cliffs, NJ, 1978.

[**Lampson et al. 76**] Lampson, B. W., Sturgis, H. E., 'Reflections on an Operating System Design,' *Communications of the ACM*, vol. 19, no. 5, May, 1976.

[**Lazowska et al. 81**] Lazowska, E., Levy, H., Almes, G., Fischer, M., Fowler, R., Vestal, S., 'The Architecture of the Eden System,' *Proceedings of the 8th ACM Symposium on Operating System Principles*, December, 1981.

[**Liskov et al. 82**] Liskov, B., Scheifler, R., 'Guardians and Actions: Linguistic Support for Robust, Distributed Programs,' *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages*, January, 1982.

[**Lockman et al. 82**] Lockman, A. and Minsky, N., 'Unidirectional Transport of Rights and Take-Grant Control', *IEEE Transactions on Software Engineering*, Vol. SE-8, no. 6, November, 1982.

[**Needham et al. 78**] Needham, R.M. and Schroeder, M.D., 'Using Encryption for Authentication in LArge Networks of Computers', *Communications of the ACM*, Vol 21, No. 12, Dec 1978.

[**Parnas 72**] Parnas, D., 'On the Criteria to be Used in Decomposing Systems into Modules,' *Communications of the ACM*, vol. 15, no. 12, December, 1972.

[**Pashtan 82**] Pashtan, A., 'Object Oriented Operating Systems: An Emerging Design Methodology,' *Proceedings of the National ACM Conference*, October, 1982.

[**Ramamritham et al. 83**] Ramamritham, K., Vinter, S. T., Stemple, D., 'Primitives for Accessing Protected Objects,' *Proceedings of the 3rd Symposium on Reliability in Distributed Software and Database Systems*, October, 1983.

[**Ramamritham et al. 85**] Ramamritham, Stemple, D., Vinter, S. T., 'Decentralized Access Control in a Distributed System,' *Proceedings of the 5th International conference on Distributed Computing Systems,* May 1985.

[**Ramamritham et al. 86**] Ramamritham, K., Briggs, D., Stemple, D., Vinter, S. T., 'Privilege Transfer and Revocation in a Port-Based System,' to appear in *IEEE Transactions on Software Engineering*, 1986.

[**Randell 78**] Randell, B., 'Reliable Computing Systems,' *Operating Systems, An Advanced Course*, Lecture Notes in Computer Science, by Springer-Verlag, editors Bayer, Graham, and Seegmuller, 1978.

[**Rashid et al. 81**] Rashid, R., Robertson, G., 'Accent: A Communication Oriented Network Operating System Kernel,' Carnegie-Mellon University Technical Report, April, 1981.

[Ritchie et al. 74] Ritchie, D. and Thompson, K., 'The UNIX Time-Sharing System,' *Communications of the ACM*, vol. 17, no. 7, July, 1974.

[Saltzer 74] Saltzer, J. H., 'Protection and the Control of Information Sharing in MULTICS,' *Communications of the ACM*, vol. 17, no. 7, July, 1974.

[Saltzer et al. 81] Saltzer, J.H., D.P. Reed, D.D. Clark. 1981. End-to-end arguments in System Design, *Second International Conference on Distributed Computing Systems*, (April).

[Schroeder et al. 72] Schroeder, M., Saltzer, J., 'A Hardware Architecture for Implementing Protection Rings,' *Communications on the ACM*, vol. 15, no. 3, March, 1972.

[Snyder 77] Snyder, L. 'On the Synthesis and Analysis of Protection Systems', *Proc. Sixth Symposium Operating System Principles*, Nov 1977, pp 141-150.

[Solomon et al. 79] Solomon, M. H., Finkel, R. A., 'The Roscoe Distributed Operating System,' *Proceedings of the 7th ACM Symposium on Operating System Principles*, March, 1979.

[Stemple et al. 83] Stemple, D., Ramamritham, K., Vinter, S., 'Operating System Support for Abstract Database Types,' *Proceedings of the 2nd International Conference on Databases*, September, 1983.

[Stemple et al. 85] Stemple, D., Vinter, S., Ramamritham, K., 'Dynamic Control of Module Interconnections', submitted to *IEEE Software*, August 1985.

[Stemple et al. 86] Stemple, D., Vinter, S., Ramamritham, K., 'Functional Addressing in Gutenberg: Interprocess Communication Without Process Identifiers,' to appear in *IEEE Transactions on Software Engineering*, 1986.

[Strom et al. 83] Strom, R., Yemini, S., 'NIL: An Integrated Language and System for distributed Programming,' *Proceedings of SIGPLAN '83, Symposium on Programming Languages,* August, 1983.

[Sturgis et al. 80] Sturgis, H.E., Mitchel,J.G., and Israel, J. 'Issues in the Design and Use of a Distributed File System', *Op Sys Rev* 14(3), pp 55-69, July 1980.

[Vinter et al. 83] Vinter, S. T., Ramamritham, K., Stemple, D., 'Protecting Objects Through the Use of Ports,' *Proceedings of the Pheonix Conference on Computers and Communication*, March, 1983.

[**Vinter 83**] Vinter, S. T., 'Communication and Protection in a Distributed Operating System,' Master's Thesis, University of Massachusetts, August, 1983.

[**Vinter 85**] Vinter, S. T., 'A Protection Oriented Distributed Kernel,' Ph.D. Thesis, University of Massachusetts, August 1985.

[**Wulf et al. 74**] 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.