The Gutenberg Operating System Kernel

[1]

Panayiotis Chrysanthis, Krithi Ramamritham, David Stemple, and Stephen Vinter [2]

Department of Computer and Information Science
University of Massachusetts Amherst MA. 01003

ABSTRACT

The Gutenberg system is a port-based, object-oriented operating system kernel designed to facilitate the design and structuring of distributed systems. This is achieved by providing primitives for controlling process interconnections and thereby controlling access to shared resources. Only shared resources are viewed as protected objects. 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. Capabilities to create ports for requesting operations are contained in the *capability directory* which is a kernel object. At any time, each process is associated with a single subdirectory of the capability directory designated as its *active directory*.

This paper describes the design philosophy and the structure of the Gutenberg kernel. First, it discusses the principles and motivations behind the Gutenberg design. Then, it presents the structure, contents and operations of kernel objects and discusses their use in structuring access to protected user-defined objects. Also discussed are the salient aspects of the distributed Gutenberg kernel. A brief comparison with other related systems is given.

1 INTRODUCTION

Modern distributed operating systems must be designed to facilitate structuring distributed computations in an understandable and reliable manner that is suitable for validation. Experience in programming languages and in controlling complexity of large software systems has shown that the strongly typed object paradigm and its associated information hiding can be used to produce manageable and understandable systems. In large distributed systems, lack of a corresponding clean, well-structured interprocess communication paradigm leads to more complexity than is required by the nature of the distribution alone. This points to the need for an efficient mechanism which structures all interprocess communications along the abstract data type lines.

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

[2]Stephen T. Vinter's current address is: BBN Laboratories, Cambridge, Mass.

This paper deals with the design of such a mechanism, the distributed Gutenberg kernel [11,12,7,8]. The Gutenberg Operating System Kernel, currently being developed at the University of Massachusetts, takes a unique approach to interprocess communication, provides multiple programming language support, and attempts to minimize overheads. The following three salient features of Gutenberg provide for the controlled establishment of communication connections which serves the goal of understandability and verifiability, and should contribute to its better performance compared to previous communication schemes:

- the adoption of *port-based communication*, whereby interprocess communication is solely by means of ports, queue-like objects which are managed by the kernel.
- the adoption of *non-uniform object-orientation*, whereby only interprocess communication, but not module interconnections within a process, are structured and controlled by means of capabilities; and
- the use of a *capability directory*, which expresses all potential process interconnections in the system and is a distributed persistent object, maintained and manipulated only by the kernel.

Principles underlying the Gutenberg Kernel. The Gutenberg system evolved from the following series of design decisions concerning the nature of communication and protection in the system.

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, e.g. Hydra [16] and iMAX [4]. 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. 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. 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 of other processes through the use of kernel primitives [11]. 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 [12]. 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. Being physically distributed, it 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. 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.

The rest of the paper is structured as follows: Section 2 introduces the structure and the contents of the kernel objects. User-defined objects, type creation, and manager instantiation are the subject of section 3. Issues related to the dynamic control of interprocess communication are also discussed in section 3. The distributed kernel is examined in section 4. Section 5 contains a brief comparison of Gutenberg with related systems. We conclude with a section summarizing our approach and future work.

2 GUTENBERG KERNEL OBJECTS

The Gutenberg kernel itself is structured as a set of cooperative abstract data type managers. Furthermore, the kernel is viewed by the processes as an abstract data type manager of kernel objects with the kernel primitives as the corresponding abstract data type operations. There are four types of kernel objects: *processes*, *ports*, *capability directory* and *transient capabilities*.

2.1 Processes

A process is an independently schedulable unit of computation with the ability to communicate with other processes. Each process is represented by a unique *process control block*, abbreviated *PCB*, which resides within the kernel address space.

Processes can communicate only 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.

2.2 Ports

A port is a kernel object that processes manipulate by invoking kernel primitives. It is a communication channel between a pair of processes in one-to-one topology connecting just one pair of processes at a time. Typically, 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.*

Therefore, 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. 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 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

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

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

status of the client and server processes and the owner of the port. Ownership represents the privilege to destroy the port. The creator of a 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 functionality of port primitives.

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. The sender 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.

EXAMINE examines messages on the port without removing them.

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 [3].

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 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.

2.3 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 [10] 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 than 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.

2.3.1 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 other cd-nodes through capabilities. The same cd-node may be linked to several cd-nodes under possibly different user-specified names. All capabilities pointing 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 a sample capability directory.

The capability directory may contain two kinds of cd-nodes: *subdirectories* and *manager definitions* (see figure 3). Note that an *asterisk* next to attribute names used in the figures designates that the attribute cannot be modified by the users but is maintained by the kernel.

A subdirectory is a list of capabilities. 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.

[3] A scheme for revoking transferred capabilities anytime after the transfer is discussed in [8].

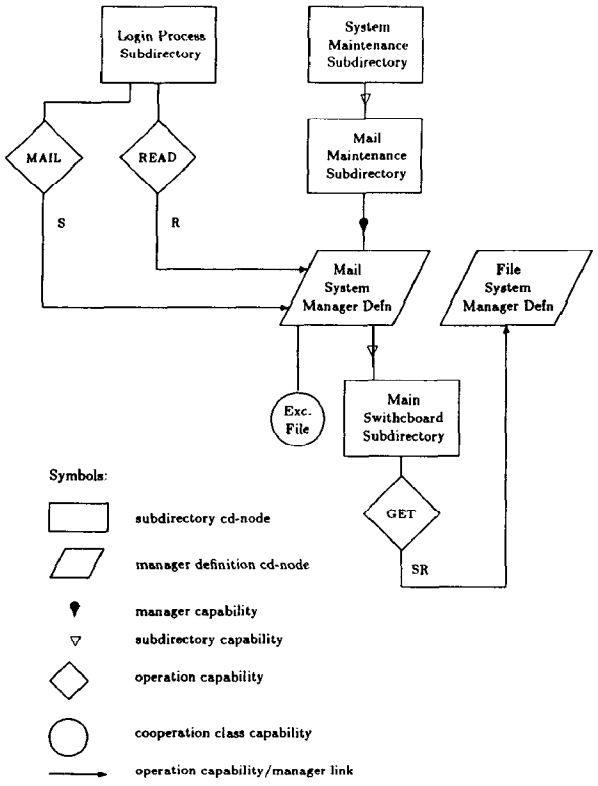

Figure 2: Example of Capability Directory Segment
This is a sample capability directory of a basic mail system. The Mail Maintenance Directory which contains the Mail System Manager Definition is created by the creator of the system and stored in the System Maintenance Directory. The two operations defined on the Mail System, namely the Mail and Read, are contained in the login process subdirectory. Each user gets capabilities for these operations during the login procedure. In the default subdirectory of the Mail System, operations on the File Manager exist that permit access to files.

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. 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 3.2) 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 other component 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.

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

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

subdirectory id* system-wide unique identifier of the subdirectory.

use count* the number of stable and transient subdirectory capabilities (including those in manager definition cd-nodes) pointing to this cd-node.

active directory count* the number of processes having this cd-node as their active directory.

Manager Definition Node Provides information necessary for instantiating a manager process. The manager definition cd-node has nine attributes.

manager id* system-wide unique identifier of the manager 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.

initial process image a privilege (represented by a cooperation class capability) for a file containing the object code to be executed when the process is initiated.

manager dependency indicates whether the existence of a manager process is dependent on the existence of ports connected to it.

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.

port descriptors is the list of (port type, generic operation name) pairs for operation capabilities that may be linked to this manager cd-node. The port type specifies whether it is a Send, Receive or Send-Receive port as well as the format of the arguments that may be passed over the port as part of a message or request-details.

manager use count* the number of manager capabilities pointing to this cd-node.

operation use count* the number of operation capabilities linked to this cd-node.

port use count* the number of ports associated with this cd-node.

Figure 3: Attributes of Capability Directory Node

Operation Capability Provides the privilege to create ports. 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.
- **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.
- **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).
- **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.
- **cooperation class(es)** restrict how the port will be connected to a manager process.
- **manager id*** a pointer, global name, to the manager to which this operation capability is linked.

The capcaps of the operation capability are: COPY, TRANSFER, REGISTER, REMOVE, HOLD, MERGE, MODIFY-CAP, MODIFY-CAPCAP, and VIEW-CAP.

Figure 4: Attributes of an Operation Capability

the other kernel primitives that manipulate the capability itself.

The capcaps determine how the capability may be modified and used. 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).

2.3.2 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 4) represents the privilege to create a port for use in 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. 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 currently supported by the manager; this is checked by examining whether the operation generic name is part of the port descriptors in the manager definition.

The primary kernel primitive in the manager definition capability (figure 5) is **Create-operation**, which is used to create operation capabilities, linked to the manager definition 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 6) 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*. When a subdirectory capability is exercised to make a subdirectory active, the subdirectory rights override the capcaps of each individual capability in the subdirectory, and this further restricts the use of the capabilities registered in the subdirectory. This restriction during the changing of the active directory, referred to as *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

Manager Definition Capability Provides the privilege to create operation capabilities. 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.
- **cooperation class(es)** restrict who may create operation capabilities linked to the definition manager. Possession of one of the specified cooperation class capabilities is required when exercising the **Create-operation** primitive.
- **manager id*** a pointer, global name, to the manager definition cd-node corresponding to this 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.

Figure 5: Attributes of a Manager Definition Capability

Subdirectory Capability Provides the privilege to change the process' active directory. The attributes are:

subdirectory name user-specified name of the subdirectory cd-node which becomes the process' active directory when the **change-directory** privilege is exercised. This name also serves to identify the subdirectory capability.

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.

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.

subdirectory id* a pointer, global name, to the subdirectory cd-node corresponding to this 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.

Figure 6: Attributes of a Subdirectory Capability

Cooperation Class Provides a wild card privilege that may be merged with any other capability type. It has two attributes:

class name user-specified name used to identify the cooperation class in the current protection domain.

class id* system wide unique identification of cooperation class.

The capcaps of the cooperation class capability are:
COPY, TRANSFER, HOLD, REGISTER, REMOVE, MERGE, VIEW-CAP, MODIFY-CAP, and MODIFY-CAPCAP

Figure 7: Attributes of Cooperation Class Capability

initial and goal subdirectory, the mechanism for switching to a new domain becomes simple, and the security of the system is not compromised.

The fourth type of capability, the cooperation class capability, represents the privilege of a process to participate in a cooperative activity identified by a unique identifier, the *class id* (figure 7). 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. New cooperation class capabilities can be created, on request, by the kernel.

2.4 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, and therefore 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 existence of the process 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.

A transient capability comes into existence when a process moves or copies a capability from its active directory, into its c-list (using a primitive called HOLD), when it receives the capability from another process via a port, when it creates a capability by invoking the proper create primitive, or when 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. A process moves 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 case, the ownership of the port is transferred along with the port capability.

The only capcap associated with a port capability is *transfer*. 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 enabled. 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 a hierarchy of object managers, features which can improve the performance, flexibility and reliability of 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 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 *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 must be allowed by the rights of the active directory of the invoking process.

The eleven generic kernel primitives are: *Create*, *Register*, *Register-C*, *Remove*, *Hold*, *Hold-C*, *Drop*, *View*, *Modify*, *Merge* and *Merge-C*. Here is a brief description of their functionality (Details concerning the primitives may be found in [3]).

Create These primitives create a capability. **Create-port**, and **Create-operation** are instances of this generic primitive. A **Create** always creates a transient capability which may then be registered or transferred with all or part of its privileges retained. Creating an operation capability requires a manager capability, and creating a port capability requires an operation capability. Other **Create** primitives require no privilege.

Register and Register-C 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. 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.

Remove These primitives delete a stable capability from the active directory.

Drop These primitives delete a transient capability from the c-list.

Hold and Hold-C 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. The purpose of these primitives is to allow a process to retain a capability from its active directory when it changes its active directory to another subdirectory.

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.

Modify These primitives allow a process to modify an existing transient or stable capability, or a cd-node.

Merge and Merge-C 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).

As has previously been discussed, the manager definition and subdirectory capabilities are slightly different from 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. 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 it has requested. 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 as we discuss below.

3 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.

3.1 Type Creation

User-defined types in Gutenberg are represented by manager definition cd-nodes (figure 3). 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.

3.2 Manager Initiation Protocols

The manager initiation protocol specified when creating a manager definition determines the manner in which ports are connected to the manager processes instantiated from the definition. 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 be-

ing 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.

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. For a more detailed description of manager initiation protocols, see [12].

3.3 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.

In Gutenberg, process interconnections 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; 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 any number of ports to access an object (identified by the cooperation class).

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. 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. These two mechanisms of capability transfer are distinguished by the semantics of the transferring. The transfer by SEND is permanent, whereas the 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. For more detailed discussion of the privilege transferring mechanisms as well as their implications see [8].

4 DISTRIBUTED GUTENBERG KERNEL

An instance of the Gutenberg kernel is running on each site in the system. From the objects that the kernel maintains only the capability directory needs to be distributed. The other three objects, namely processes, ports and transient capabilities, are naturally distributed since processes always execute on a single site, and ports and transient capabilities can only be used by the process which possesses them.

The capability directory is partitioned and replicated to ensure availability and reliability. Manager definitions are replicated only at the sites in which manager processes from these can possibly be instantiated. Subdirectories are replicated in locations where they are expected to be used and in a number of other locations in a manner that meet the resiliency requirement and balance the distribution of local directory storage space. The site where a cd-node is created has a copy of that cd-node and also keeps track of which sites have copies of that cd-node.

The set of capabilities and cd-nodes from the capability directory that resides on a site forms that site's *local directory*. The site's local directory dynamically expands and contracts to accommodate the needs of any process executing on the site. *Dynamic adjustments* pertain only to the migration of subdirectories; modifications to manager definitions are expected to be relatively rare to justify their migration. A subdiretcory is migrated only when processes make repetitive use of the capabilties located there, justifying the move. Even then, only (lockable) portions of the subdirectory are copied in a *lazy copy* fashion.

Synchronization of access to components of the capability directory is achieved with the use of two locking-based concurrency control schemes with the granularity of locks being on portions of cd-nodes. For example, a subdirectory is associated with three locks; one is used for the set of operation and cooperation class capabilities; the second for subdirectory capabilities; and the third for manager definition capabilities.

Two concurrency control schemes used are the *primary two-phase locking scheme* [2] where a single copy is designated as the site to gain all locks, and the *basic two-phase locking scheme* [2] where to lock a cd-node for reading it is necessary to lock only the local copy whereas a write requires obtaining the write lock on all replicates of the cd-node. Which scheme is used for a specific cd-node depends on the likelihood of its predominant access by processes on one site or on multiple sites. A *dynamic recategorization* allows the kernel to adapt to the system behaviour by changing the concurrency control mechanism for a cd-node in response to the nature of its use.

A two-phase commit protocol is used to achieve a reliable

distributed commitment and ensure atomicity. For a more detailed discussion of issues involved in the distribution and the approach adopted see [7].

5 A COMPARISON WITH RELATED SYSTEMS

A large part of the improvement in programming languages has come from the promotion of data and procedural abstraction as a major tool for structuring modules. This led to the adoption of abstraction and encapsulation mechanisms in Gutenberg, as in many other systems, e.g. Argus [6], NIL [13]. While in these two systems, the mechanism for dynamic module interconnection control is built within a programming language, the approach taken in Gutenberg separates process interconnection control from programming languages. Gutenberg supports the dynamic control of process interconnections through the use of capabilities. However, it is different from the other protection-oriented operating systems such as Hydra [16] and iMAX [4], in that it adopts a non-uniform object-orientation.

A few systems provide port-based communication facilities using functional addressing [12], but none ties protection so closely to communication as Gutenberg does. The Accent system is port-based and supports asynchronous communication with process transparency [9]. Communication in Gutenberg is also similar to the mechanisms used in Intel iAPX-432 [4], DEMOS [1] and NIL [13]. 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 is not supported directly, as in Gutenberg.

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 [5] 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 [5]. 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.

6 CONCLUSION

The Gutenberg system is a novel attempt to facilitate the design and structuring of distributed computations in an understandable and reliable manner that is suitable for validation. The crux of the Gutenberg approach is the use of port-based communication, non-uniform object-orientation and decentralized access authorization using ports and the capability directory. This nonprocedural directory provides an abstract view of the functional building blocks of a large system of distributed cooperating modules and should serve the goal of understandability and verifiability.

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 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.

References

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

[2] Bernstein, P., Goodman, N., 'Concurrency Control in Distributed Database Systems,' *ACM Computer Surveys*, vol. 13, no. 2, June 1983.

[3] Chrysanthis, P.K., Ramamritham, K., Stemple, D.W., Vinter, S.T., 'The Gutenberg Operating System Kernel,' Dept. of Computer and Information Science Technical Report 86-06, University of Massachusetts, February, 1986.

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

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

[6] 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.

[7] 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.

[8] Ramamritham, K., Briggs, D., Stemple, D., Vinter, S. T., 'Privilege Transfer and Revocation in a Port-Based System,' *IEEE Transactions on Software Engineering*, vol. SE-12, no. 5, May 1986.

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

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

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

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

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

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

[15] Vinter, S. T., Ramamritham, K., Stemple, D., 'Recoverable Communicating Actions,' *Proceedings of the fifth International Conference on Distributed Computing Systems*, May 1986.

[16] 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.