The Gutenberg Experiment: Testing Design Principles as Hypotheses

P.K. Chrysanthis,
K. Ramamritham, and D.W. Stemple
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

COINS Technical Report 89-36

The Gutenberg Experiment: Testing Design Principles as Hypotheses *

Panayiotis K. Chrysanthis
Krithi Ramamritham
David W. Stemple

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

Abstract

The Gutenberg operating system has been designed to facilitate the development of manageable and understandable systems comprising many distributed, cooperating modules. Gutenberg is port-based with the creation of types controlled by a system wide directory of capabilities that control the interconnection of processes. Port creation capabilities are labelled with the names of operations that processes can request from their external environment. This imposes an abstract data type view on all requests of a process to its environment and is meant to make complex distributed systems more coherent and understandable than is common in other approaches.

The goal of the Gutenberg experiment was to test the expressiveness and implementation difficulty of the Gutenberg paradigm. To this end, a prototype system was built and applications for it were implemented. The design and implementation efforts were opportunities for refining the design, an effort that was itself experimental in nature. The refinement of the design was organized as a series of hypotheses that certain principles would lead to better quality, followed by redesigning in accordance with the implications of the hypotheses, and evaluating the results. In general, we found that the application of symmetry and uniformity principles improved both the expressiveness and ease of implementation of the Gutenberg functions.

*This material is based upon work supported by the National Science Foundation under grant CCR-8500332.

1 Introduction

In large distributed systems, the lack of a clean, well-structured paradigm for processes and their interactions leads to more complexity than is required by the nature of the distribution alone. Gutenberg attempts to solve this problem by taking a unique approach to interprocess communication.

Gutenberg is a port-based, capability-based distributed operating system kernel designed and implemented at the University of Massachusetts. It presents to its users with an *object virtual machine* in which all interactions among objects are structured along abstract data type lines. The purpose of Gutenberg is to facilitate the development of manageable and understandable systems made up of many distributed cooperating modules.

During the first phase of the project, started in 1983, the Gutenberg kernel evolved from a series of design decisions concerning the nature of communication and protection in distributed systems [Rama83, Rama85, Vinter85, Rama86, Stemple86, Chrys86]. The implementation of the Gutenberg Kernel was taken up in the second phase. This implementation, called the Gutenberg experiment, started in October 1985. This paper reports on the knowledge gained from this experiment.

The two broad goals of the Gutenberg Experiment were: to evaluate the *expresiveness* of the Gutenberg kernel, i.e., the nature of applications that can be built using the Gutenberg primitives, and its *programmability*, i.e., the ease with which these applications can be built, and to discover difficulties in implementing the logical level of the Gutenberg Kernel. We were interested in understanding the kernel structures novel to Gutenberg, namely, the *Interconnection Schema* and *typed ports*. The former is a persistent structure that specifies potential process interconnections. Typed ports are used to restrict the use of ports to the specific purpose for which they have been created. We were not interested in placing emphasis on areas where Gutenberg was not attempting to break new ground. Thus, issues related to process and resource management, such as memory management, were not of primary importance to us. However, we were interested in the performance of the system and the cost of interprocess communication but not in expressing them in terms of time. We were interested in the communication overheads resulting from Gutenberg functionality. This can be expressed by the number of operations on the various Gutenberg structures and in particular, number of memory allocations and copies, needed to construct and deliver a message. Cost in time may

be useful for making "raw" comparisons between systems but it fails to provide any information for identifying bottlenecks. The cost in terms of the number of operations not only provides the information for identifying bottlenecks but it also serves as a hint of what is needed to improve performance.

Based on these goals and since we were not interested in a testbed for gathering numbers, we decided to implement a prototype Gutenberg kernel on top of an existing operating system. This prototype Gutenberg is the subject of section 3. A review of the Gutenberg environment and the principles underlying its design are given in section 2. The evolution of the prototype, and particularly of the Gutenberg primitives, according to a series of hypotheses is discussed in section 4. A critique of the Gutenberg system based on the knowledge gained so far from the experiment is presented in section 5. Section 6 compares Gutenberg with related systems.

2 Principles Underlying Gutenberg

Object Managers request operations through Ports

In Gutenberg, a distributed system is a group of *object managers* that execute asynchronously and concurrently, interacting cooperatively to perform a task. An object represents a sharable resource accessible via specific operations. Object managers synchronize the operations on objects and are the only subjects able to directly manipulate the state of objects.

Since object managers do not share address space, they communicate only through explicit message exchange over communication channels called *ports*. Objects are instances of abstract data types, and hence, *inter*manager communication in Gutenberg is always in terms of requests for abstract data type operations. While Gutenberg enforces an object-oriented view on all intermanager communication, it does not enforce this view upon the programs implementing an object manager. The organization of *intra*manager communication depends on the programming language used to build the object manager and can be object-oriented or not.

Ports between object managers are created based on the need to provide or request a service. Thus, Gutenberg has adopted the client/server model for intermanager communication in which the user of a port, called the client, sends a request for an operation to the object manager, which then performs the operation and may send back a reply. Object managers can be clients as well as servers. Furthermore, object manager

interconnections can be dynamically changed by transferring ports over other ports.

In order to restrict the use of ports to the functionality for which they have been created, a port is *typed* with respect to its directionality and message contents.

Functional Addressing: Communication without Process Indentifiers

A port is established using *functional addressing*. A client creates a port by naming the service (the operation) it would like to request using the port rather than by identifying the server object manager. The advantage of this strategy is that it supports service transparency, allowing for dynamic object re-implementation and/or relocation, an important property for distributed systems. The client object manager does not have to know the identity of the server object manager, or whether that object manager executes on a local or remote machine. The server object manager does not even have to be active prior to the creation of the port.

Object manager activation and deactivation in Gutenberg are sideffects of port operations. There are no system calls to explicitly activate or deactivate object managers as in systems which support the notion of processes at logical level. In this sense, Gutenberg is a *processless* system.

Port-based Access Control

The kernel enforces a port-based control of access to user-defined objects. The manager of object A can create a port for requesting an operation on an object B only if it has the *capability* to execute that operation. After port creation, the only check that needs to be made when the object manager A requests access to the remote object B via this port is whether object manager A has the capability to access that port. In a distributed system, the kernel components local to the node in which object manager A resides carry out this check, thus making it a *local* check.

The Interconnection Schema

The capabilities for creating ports are stored in the *Interconnection Schema*, a persistent, distributed object managed by the kernel. The Interconnection Schema expresses all the potential object interconnections achievable by programs running under the kernel's control. This means that the Interconnection Schema stores information about the organization of applications runnable under the Gutenberg kernel at any given time.

Besides enforcing interconnection structure, the Interconnection Schema also supplies the kernel with the information needed to locate, and if necessary activate, the

server of a port. In this way, the Interconnection Schema acts as a *name server*. It also contains the definitions of object managers (see section 3.2.2), and the rules for activating the object managers. New managers are introduced into the system by storing their definitions in the Interconnection Schema.

The interconnection schema has a directory-like structure. At any time, each object manager in the system is associated with a segment of the Interconnection Schema, designated as its *active directory*, and with its transient capabilities stored in its *capability list*, abbreviated *c-list*. Each object manager is associated with a single c-list that cannot be shared. The *transient* capabilities persist only as long as an owning object manager is active. Gutenberg supports dynamic access control by allowing object managers to change their active directory thereby acquiring new capabilities, or by transferring capabilities over ports.

Capabilities in Gutenberg

Gutenberg capabilities are unlike capabilities in other systems where capabilities contain the identities of the objects on which they allow operations. The reason for this difference is that Gutenberg capabilities are capabilities that allow operations on *kernel objects*, namely the entities in the interconnection schema and ports. These capabilities are used to create privileges for operating on user-defined objects. The identification of user-defined objects is accomplished by using a capability unique to Gutenberg: the *cooperation class* [Stemple86]. Cooperation classes are generic capabilities that can be attached to distributed activities in order to provide the basis for coordinating cooperation. Communication among objects/managers that have no explicit means of addressing each other is one of the major activities cooperation classes facilitate.

More details about ports and the Interconnection Schema, and the kernel implemented operations for manipulating them, called *primitives*, are given in the next section.

3 The Gutenberg Prototype

In this section, we first provide a brief history of the Gutenberg experiemnt to place our work todate in perspective. The structure of the Gutenberg system is discussed in 3.2. Kernel objects, namely the ports, the capabilities and the Interconnection Schema, are presented in 3.3. The kernel primitives that manipulate the kernel objects, are detailed in 3.4.

3.1 Brief History

The goals of the experiment and a desire to accelerate the development of the prototype Gutenberg system, moved us to layer the implementation on an existing kernel. UNIX[1] [Ritchie74] was chosen because of the diversity of existing tools and facilities, and for portability reasons. The presence in our environment of a number of μVAXes running Ultrix[2] and supporting X Windows[3] strongly influenced this decision. Multiple windows allowed us to easily build a visual multiuser environment in which each user is handled by a Gutenberg object manager (*login manager*) associated with a window. Furthermore, they would be of a great help while debugging and testing the kernel.

The other alternatives, that did not have these advantages, which we considered were to fully implement the Gutenberg kernel on a network of Personal Computers or to implement the kernel on top of VMS[4] on a collection of VAXes.

During the first 18 months of the experiment, we built and refined the first version of the prototype Gutenberg kernel executing on a single μVax [Chrys87]. The kernel is written entirely in C. Subsequently, it was ported to a Sequent multiprocessor running Dynix[5]. Porting the kernel to a different machine, and in particular to a multiprocessor, helped us in two ways: first, to clearly identify the machine-dependent parts of our kernel implementation, and second, to detect race conditions in both our kernel and X Window interface which otherwise would have been hard to detect. We also learned first hand that compatibility between UNIX implementations is not guaranteed. Although we did not experience any problems in compiling and booting the kernel on the Sequent, the kernel did not execute correctly and in some cases crashed. We discovered that one of the reasons for this was that some Ultrix functions were implemented as macros in Dynix causing sideffects. Also, for some functions, the returned value was of a different type (real instead of integer) or format in the two implementations.

The experience gained from building the first Gutenberg prototype kernel, building applications within the Gutenberg environment, and experimenting with the porting of the kernel, led us to modify both the Gutenberg implementation and the user-level interface during the third year of the experiment. This second version of the kernel contained added functionality while maintaining the simplicity of the interface and

[1] UNIX is a trademark of Bell Laboratories
[2] DEC's implementation of Berkley UNIX 4.2
[3] X Window System 10 developed by MIT's Project Athena
[4] VMS is a trademark of Digital Equipment Corporation (DEC).
[5] Sequent's implementation of Berkley UNIX 4.2 and AT&T System V.

implementation [Chrys88]. In fact, the modifications to the system led to a better layering and modularization and reduced its size by 30%. The kernel's execution code size is about 70 kbytes.

With the experience gained from the second version of the kernel, we very recently began the distribution of the kernel over several machines. The improvements in the second version have made distribution straightforward. The network manager which facilitates the interkernel communication and the the kernel component which implements the interkernel protocol are already operational. The network manager is structured as an object manager, outside the kernel, similar to the login object managers. The network managers allow object managers to execute on various nodes and communicate by establishing *external* ports, i.e., ports across two nodes. The design for distributing the Interconnection Schema has been completed and the implementation is to be undertaken. The distributed kernel currently executes on a collection of four μVAXes.

3.2 System Organization

The basic system organization has been invariant through the three phases of the Gutenberg experiment (figure 1). The Gutenberg prototype kernel is implemented on top of the Ultrix operating system. The kernel is implemented as an Ultrix process. The object managers are implemented as separate Ultrix processes. These processes, conforming to the philosophy of Gutenberg, do not share address space. Communication between the kernel process and other processes which implement object managers, is done through a mailbox facility. The mailbox facility is implemented using signals and the shared memory mechanism provided by the underlying Ultrix.

3.2.1 The Gutenberg Kernel

The Gutenberg kernel is organized as a set of managers (see Figure 2), each of which is responsible for some of the kernel's objects. There are four managers: Kernel Control Manager, Interconnection Schema Manager (ICS Manager), Port Manager and Process Manager.

The *kernel control manager* consists of the critical components of the kernel. These are the interrupt handler, kernel setup, checkpointing module, monitor facility, and mailbox module. The *process manager* takes care of the creation and destruction of

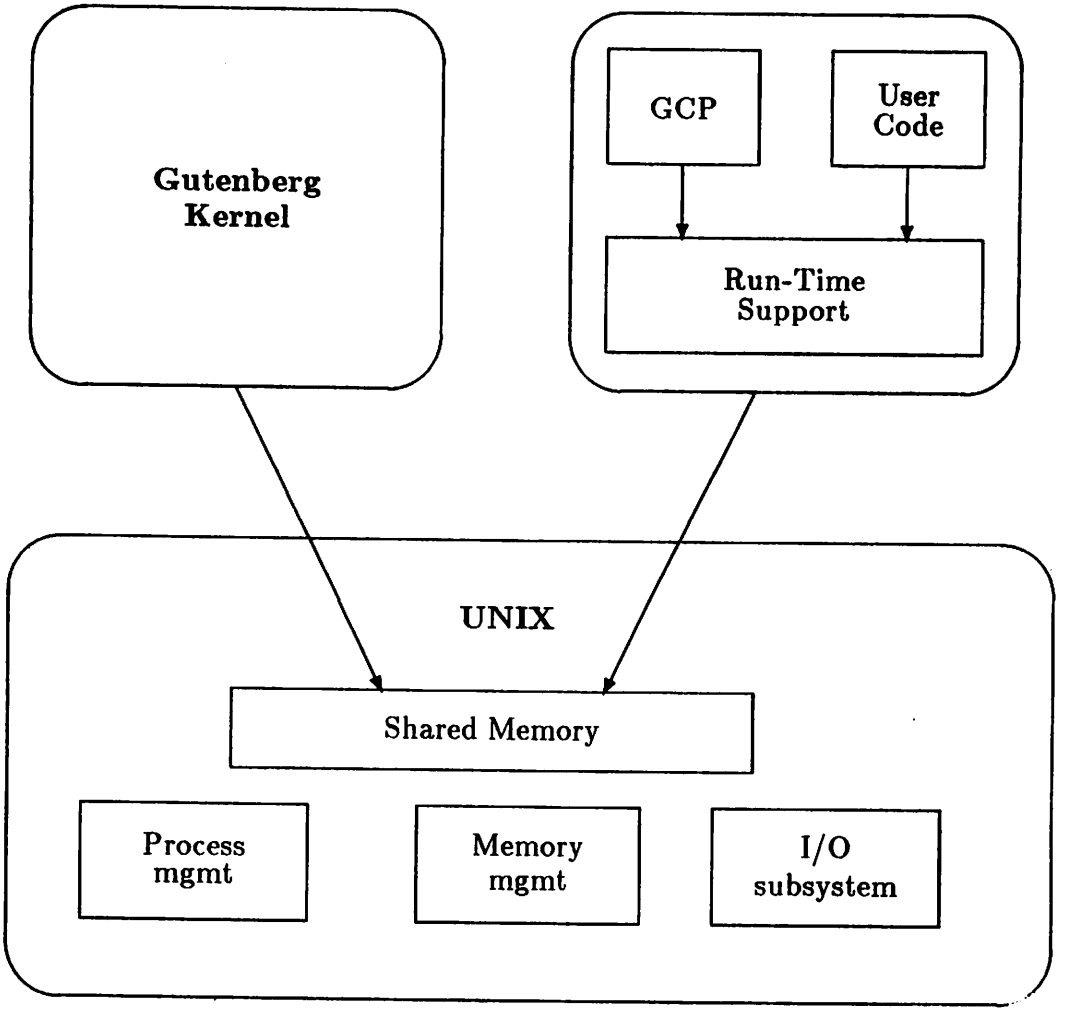

Figure 1: Gutenberg Basic System Organization

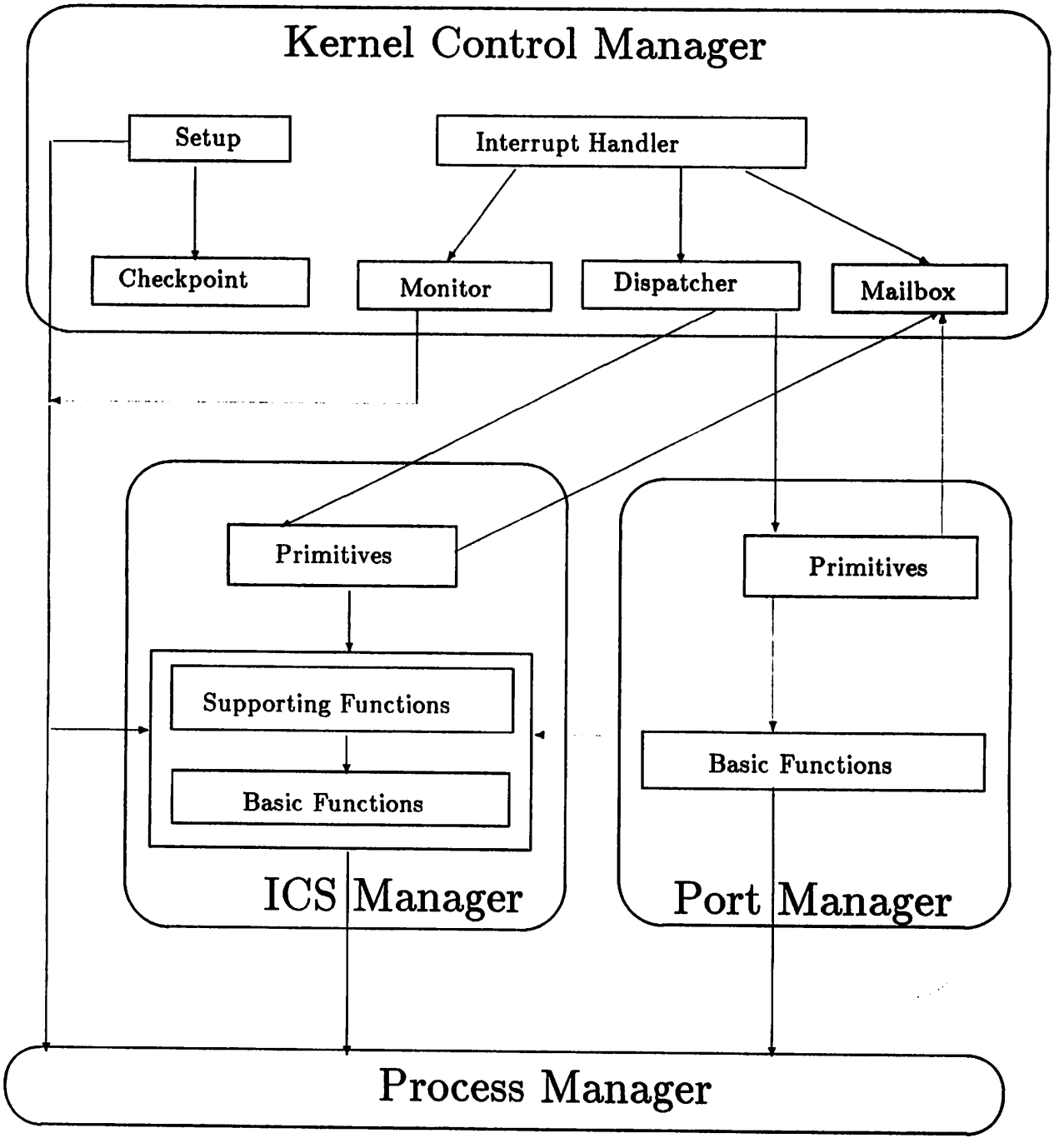

Figure 2: Gutenberg Kernel Organization

processes. The *ICS manager* provides the primitives to manipulate the Interconnection Schema. It also has a set of basic functions that manage the capabilities. These are extensively used by other managers in the system. Finally, the *port manager* provides the primitives for the port based operations.

In order to understand the control flow, here is a brief description of how the kernel works. When the system is started, the kernel setup calls the checkpoint module to read the components of the Interconnection Schema from the checkpoint files and reconstruct the Interconnection Schema. It also calls the process manager to create the processes corresponding to the login object managers. After initializing the global data structures and setting up the interrupt handlers, the kernel blocks waiting for an interrupt to arrive.

The invocation of a primitive works as follows. The invoking process places the request parameters in the mailbox and signals the kernel. On receiving a signal, the corresponding interrupt handler takes over the control. The interrupt handler polls the mailboxes of all the processes in a round-robin fashion since all processes use the same signal (Ultrix provides only two user-defined signals). If there is a new message in a mailbox, the request parameters are read and passed to the dispatcher. The dispatcher directs the request to the appropriate primitive-function in the Interconnection Schema manager or port manager module. The primitive-function in either module passes the results to the mailbox module which in turn sends them to the invoking process.

3.2.2 Object Managers

Each Gutenberg object manager is implemented by a process. The kernel instantiates a manager, as a result of the creation of the first port served by the manager. This is done by using the appropriate manager definition in the Interconnection Schema. The operation capability used in the creation of the port identifies this manager definition. The manager definition contains the *operation descriptors*, the specification of the operations that are implemented by the manager as well as the necessary information for instantiating the manager process such as a cooperation class capability for the file containing the executable image of the process. It also includes a directory capability for the *initial active directory* of all manager processes instantiated from the manager definition; the *lifetype* or *instantiation protocol*, indicating how manager processes are instantiated; and, the *interactive* flag, if set, requires the kernel to allocate an input/output device to the manager – a window in the current implementation. (The interactive flag is a

feature peculiar to the prototype kernel, and would not be present in standard manager definitions.)

All the capabilities needed by a manager for creating ports to other managers in the same application, or for changing its active directory and for moving from one directory to another within the Interconnection Schema, are stored in the manager's initial active directory.

The kernel determines the manner in which ports are connected to the manager processes instantiated from the definition, from the lifetype of the manager and the cooperation class used at port creation time. The *lifetype* specifies whether all the object instances of a type are managed by one manager or whether each object instance is managed by different manager. There are three different lifetypes in Gutenberg: *conservative*, *creative* and *class conservative*. In the case of *conservative* lifetype, a manager process is instantiated from the manager definition only if no other manager process of this type is executing in the system. If such a process already exists, the created port is attached to this process. This lifetype 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. A manager with this lifetype is informed of the object being accessed at port-creation time using a cooperation class capability. Instantiating more than one conservative manager for a type requires the creation of more than one manager definition.

Creative lifetype creates a new manager process from the manager definition for each new port created. This is useful in cases of shortlived managers owned, in effect, by the caller. That is, it 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.

Class conservative lifetype 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.

It should be clear that one of the steps in the development of any Gutenberg manager is the mapping of the design of the manager to an Abstract Data Type definition (*manager definition*) in the Interconnection Schema. Upon successful creation of a manager definition, the kernel places a *manager definition* capability, pointing at the new manager definition and containing its name, in the creating process's c-list. After

the manager definition and the corresponding manager definition capability are created, the process may use the manager definition capability to create operation capabilities needed for establishing ports for requesting the operations on the object. These operation capabilities can then be stored in the Interconnection Schema or distributed over ports to processes that should be allowed to use the type.

Although the Gutenberg design does not place any restrictions on the programming language used in coding the managers, the current implementation of Gutenberg prototype kernel supports only managers written in C.

All the routines needed for developing a manager are contained in a *software development library (sdl)*. These include all the run-time support routines required by a process to set and clean up its environment and invoke the Gutenberg primitives, their supporting functions, and the Gutenberg specific I/O routines.

Interactive managers can be built using the functions of the *Gutenberg Command Processor (gcp)*. Gcp is not part of the Gutenberg kernel, but it is a good example of a Gutenberg manager and illustrates how the Gutenberg primitives can be used. Gcp is written to support interactive users. The login object manager, upon successful login, starts up a gcp which prompts for a command. Gcp executes commands read from a terminal (window in the current implementation) or from a file (script). It recognizes two kinds of commands: those which allow the invocation of a kernel primitive to manipulate the Interconnection Schema; and those which allow the invocation of an operation on a user-defined object managed by another process.

In the sequel, the terms *object manager* and *manager process* which implements the object manager, are used interchangeably.

3.3 Gutenberg Kernel Objects

Typed ports and the Interconnection Schema are the two kernel structures unique to Gutenberg. These structures as well as their component structures constitute the Gutenberg kernel objects. Object managers can manipulate the kernel objects by invoking kernel primitives.

3.3.1 Typed Ports

A port is a communication channel between a pair of processes. At any time, only two processes have the capability to access a port for either placing messages on it or

removing messages from it. A port behaves as a queue of messages awaiting delivery. Communication through a port can be synchronous or asynchronous. The representation of the port and the details of message transmission and reception are hidden from the communicating processes. For each port created in the system, the kernel maintains a *channel control block (CCB)*.

A typed port is classified with respect to its directionality, the format of the message, and its use. The directionality of the port could be *Send*, *Receive* or *SendReceive*. Send and Receive ports are unidirectional. SendReceive ports are bidirectional, allowing the port's client to send a message and receive a response from the port's server. Typed ports are used to restrict the use of ports to the specific purpose for which they have been created.

3.3.2 Capabilities in Gutenberg and the Interconnection Schema

The Interconnection Schema is a repository of type definitions and capabilities. The Interconnection Schema is structured as a directed graph. The nodes of the graph correspond to the Interconnection Schema components, called *interconnection schema nodes*, abbreviated as *is-nodes*. The edges of the graph correspond to capabilities. Is-nodes contain various information along with capabilities that points to other is-nodes. Is-nodes are identified by user-specified names stored in the capabilities that point to them. Capabilities that reference the same is-node may have different user-specified names. All capabilities that point to an is-node have equal status. Is-nodes are unique and are *not* contained within other is-nodes. An is-node exists independently of any other is-node and disappears along with the last capability link to it, unless it is explicitly destroyed. Figure 3 shows the components of the Interconnection Schema and their attributes.

The Interconnection Schema may contain three kinds of is-nodes: *directories*, *manager definitions*, and *cooperation classes*. A directory is a collection of capabilities and serves as an organizational unit of the Interconnection Schema, similar to a file directory in a file system. On the other hand, manager definitions are the operational components of the Interconnection Schema, being Abstract Data Type definitions.

The cooperation class is-node was introduced for implementation purposes and it is *not* visible to users. It facilitates the garbage collection in the Interconnection Schema. It should be noted that the problem of garbage collection of the Interconnection Schema is identical to the problem of dangling capabilities in Gutenberg. Both problems are

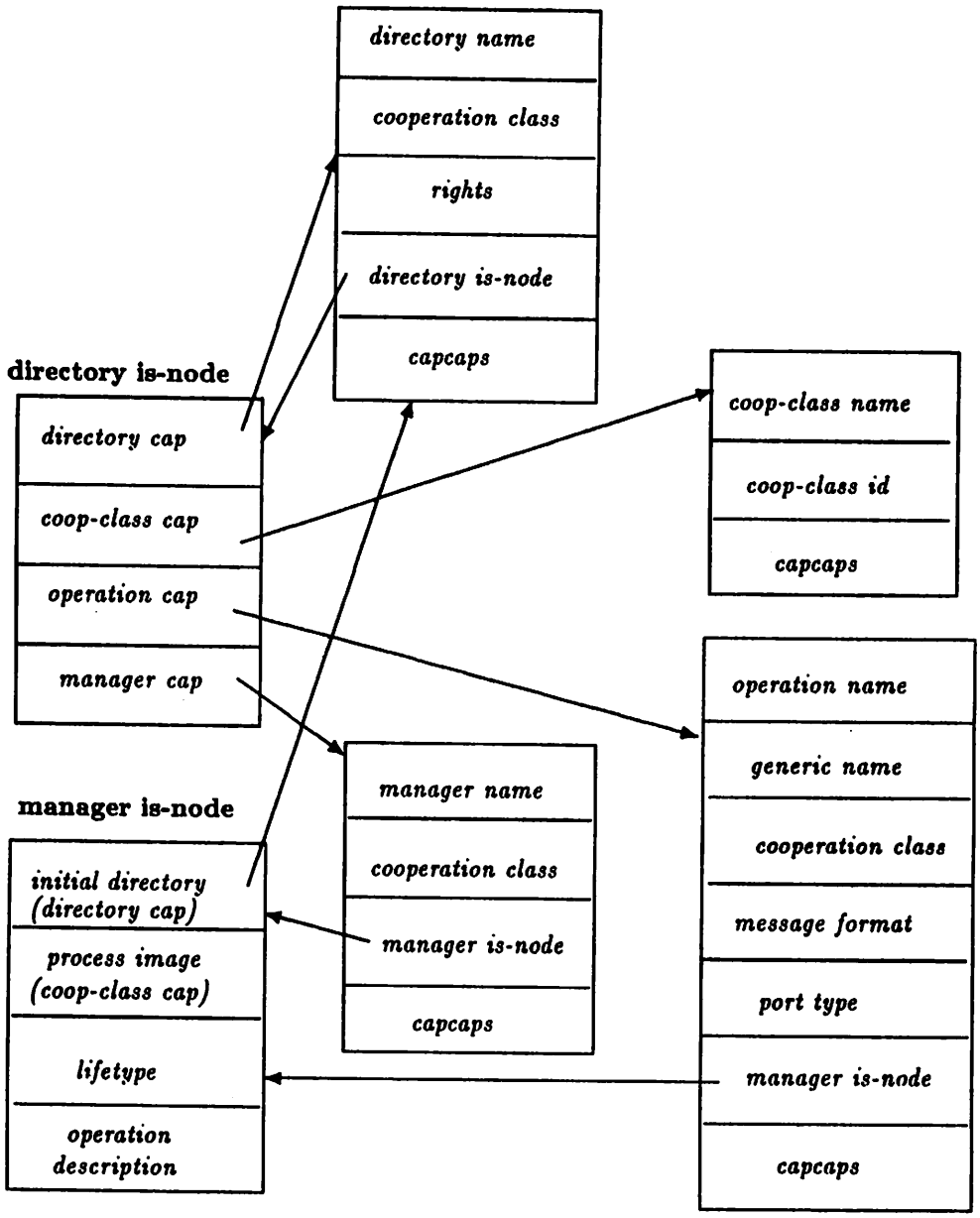

Figure 3: Logical components of the Interconnection Schema

solved by using reference counters.

Capabilities in Gutenberg consist of two parts: a list of parameters, and a list of primitives (see section 3.4 for their description) used to manipulate the capability itself. These are called *capcaps*, for capabilities on a capability. Capcaps include the privilege to *transfer* (to another process), *copy*, *register* (in the Interconnection Schema), *hold* (make transient), *merge* (with other compatible capabilities), *view*, *modify*, and *associate* the capability. Thus, the capcaps determine how a capability may be modified and used. 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.

The parameter list may include names (links or pointers) of is-nodes as well as other capabilities (most notably, the *cooperation class* capability). Links in a capability merely identify is-nodes. On the other hand, a pointer X in capability Y indicates the ability to access the object Y using the capability X. The parameter list is initialized when the capability is created by the kernel. Subsequently, some parameters can be modified by processes if they have the capability (given by a capcap) to do so.

The parameter list is intented to be used by the associated kernel primitive (*primary* primitive) that a process which possesses the capability is allowed to invoke. These primitives are listed in figure 4. The type of a capability specifies the primary primitive.

There are five different types of capabilities in the system: *directory*, *manager definition*, *operation*, *cooperation class capabilities*, and *port capabilities*. Although all five types of capabilities can be stored in a c-list, only the first four types may be stored in the Interconnection Schema. Port capabilities are inherently transient since they are *exclusive*, i.e., they cannot be shared and their existence depends upon the existence of the processes they connect. On the other hand, capabilities in the Interconnection Schema are *stable* in that their existence does not depend on the existence of any process. They can also be shared since more than one process can concurrently access the same segment of the schema.

A directory capability points to a directory is-node. This capability allows a process to change its active directory to the directory to which the capability points. An important attribute of a directory capability is a set of *directory rights*. When a directory capability is used to make a directory active, the directory rights override (mask) the capcaps of each individual capability in the directory. This further restricts

Capability Type	Primary Primitives
Directory	CHANGE-DIRECTORY
Manager Definition	CREATE-OPERATION
Operation	CREATE-PORT
Cooperation Class	ASSOCIATE
port	SEND, RECEIVE
	SENDRECEIVE, GET-DETAILS

Figure 4: Primary Primitives of Capability Types

the use of the capabilities stored in the directory.

A manager definition capability points to a manager definition is-node. This capability allows a process to create operation capabilities for requesting operations specified in the manager definition is-node.

An operation capability is linked to a manager definition. It is used to create ports for requesting the operation named in the capability on a given object managed by an object manager instantiated from the manager definition.

The cooperation class capability is not a traditional Gutenberg capability in the sense that is not associated with a specific primary primitive on an object which it protects. It represents the privilege of an object to participate in a cooperative activity and thus, it can be *associated* with any other capability type. The semantics of a cooperation class capability vary with the type of capability it is associated with.

When a cooperation class capability is associated with either a directory or a manager definition capability, it restricts the invocation of the associated primitive to the processes which possess a corresponding cooperation class capability. In this way, directories and manager definitions participate in a cooperative activity represented by the capability. In this case, the semantics of the cooperation class capability are similar to that of a password and complement the role of capcaps and directory rights.

When a cooperation class capability is associated with an operation capability, the user possesses the cooperation class capability by virtue of possessing the operation capability. The association of a cooperation class capability with an operation capability restricts the way the created port is attached to its server. When the server accepts the new port, it gets possession of the used cooperation capability. In general, a cooperation class capability can be used to identify either an instance of a manager (recall our discussion of the class conservative manager lifetype), or a particular instance of an object within a manager (a file for example). It can also be used as a synchronization

token.

The last type of capability, the port capability, allows the invocation of an appropriate port primitive, i.e., SEND, RECEIVE or SENDRECEIVE, on the port for which the capability was created. When a port is created, two port capabilities are generated, one for the client of the port and one for the server of the port. Thus, ports support *only* one-to-one communication allowing at a given time only one process to access the client-end and one pocess to acess the server-end of a port. A copy of the cooperation class capability used in the creation of the port is also generated for the server of the port.

All capabilities in the system are uniquely specified by the combination of the user-defined local name of the kernel object they protect, the user-defined local name of the cooperation class to which they belong, and their type. Two capabilities with the same specification cannot exist in a directory or a c-list. For this reason, all primitives that add a capability in a directory or c-list will fail if another capability with the same specifications already exists in the directory or c-list.

3.4 The Gutenberg Primitives

The Gutenberg primitives comprise the primitives for port based operations and those, collectively referred to as *schema primitives*, for manipulating the Interconnection Schema and c-lists. Communication between object managers and the kernel, (i.e., for invoking a kernel primitive), is not done through ports (e.g., as in MACH [Baron85]). Instead, the invoked kernel primitive is trapped by the kernel.

3.4.1 Port Primitives

The Gutenberg kernel provides twelve primitives for port-based operations. Figure 5 shows the implemented port primitives that clients and servers may use for each port type. A client invokes the CREATE-PORT primitive to create a port of a specific type for requesting an operation on an object. The client should possess the capability for that operation in order to invoke the CREATE-PORT primitive. The kernel attaches the newly created port to the appropriate server. The server does an ACCEPT-REQUEST to obtain access to the server-end port (capability). A client requests an operation over the port by invoking the appropriate primitive, i.e. SEND, RECEIVE or SENDRECEIVE, supported by the type of the port. The server can choose either to service the request by

Port Type	Client Primitives	Server Primitives
any	CREATE-PORT DESTROY REVOKE	ACCEPT-REQUEST REJECT REFUSE QUERY-PORTS
Send	SEND	RECEIVE EXAMINE
Receive	RECEIVE	SEND
SendReceive	SENDRECEIVE	GETDETAILS SEND EXAMINE

Figure 5: Implemented Port primitives used by clients and servers for each port type

invoking the appropriate port primitives, or to refuse service by invoking the REFUSE primitive. A server can use the EXAMINE primitive to decide which port to service next and which requests to refuse. A client can abort the last pending request on a port by invoking the REVOKE primitive or even destroy the port with DESTROY. A server of a port can permanently refuse to service a port by invoking REJECT to destroy of the port.

The owner of the client-end port capability can DESTROY the port; whereas the owner of the server-end port capability can REJECT the port. The creator of a port becomes the initial *owner* of the client-end port capability. The first server of a port becomes the *owner* of the server-end port capability.

The bidirectional SENDRECEIVE primitive and its receiving counterpart GETDETAILS are provided, in addition to the basic SEND and RECEIVE primitives, in order to allow for remote procedure call semantics to be implemented by a single primitive. A single primitive is better for both performance and software engineering reasons (less prone to errors than a pair of primitives). SENDRECEIVE is more robust and flexible than a remote procedure call because it supports both synchronous and asynchronous modes.

As part of the sharing mechanism supported by the Gutenberg system, a process may transfer part of its capabilities, including port capabilities, to another process through a port. The transfer may be either temporary (the semantics of a lend with the ownership retained in the case of port capabilities) with the SENDRECEIVE primitive or permanent with the SEND primitive. If the SEND primitive is used in the transfer, the

ownership of the corresponding port-end is transferred along with the port capability. Here is a short summary of the functionality of implemented port primitives:

CREATE-PORT: (a client primitive) creates a port of a specific type for requesting a specific operation.

ACCEPT-REQUEST: (a server primitive) is used to obtain access to newly created ports.

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 in which case the SEND is synchronous.

RECEIVE: requests the next message from the port. If there is no message on the port, the caller may elect to either block, or execute concurrently with the servicing of the request.

SENDRECEIVE: (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.

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

DESTROY-PORT: (a client primitive) destroys a port. The caller must be the owner of client-end of the port.

REJECT: (server primitive) rejects service to a client by destroying the port. The caller must be the owner of the server-end of the port.

REVOKE: (a client primitive) revokes the last request over a Send, Receive or SendReceive port up to the receipt of the request.

REFUSE: (a server primitive) rejects a client's request for service and notifies the client by setting a status.

QUERY-PORTS: (a server primitive) is used to query a set of existing ports to see if new messages have arrived.

EXAMINE: (a server primitive) examines messages on the port without removing them.

3.4.2 Schema Primitives

The kernel provides twelve generic primitives for manipulating the capabilities both within a c-list and an active directory. Recall that certain capcaps and rights are associated with each primitive. These must be active in the capability on which the primitive is invoked. Also, invocation of the primitive must be allowed by the rights of the active directory of the invoking process.

Here is a brief description of the implemented generic primitives.

CREATE: primitives create a capability. CREATE-OPERATION, and CREATE-PORT are instances of this generic primitive. A CREATE always creates a transient capability which may then be registered in the Interconnection Schema (see REGISTER primitive discussed next) 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. CREATE-DIRECTORY and CREATE-MANAGER primitives require no privilege.

REGISTER: makes a transient capability stable. 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 will share access to a directory. This primitive can be either *constructive* in that it makes a copy of the transient capability or *destructive* in that it removes the transient capability from the caller's c-list.

HOLD: makes a stable capability, transient. 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 directory. This primitive can be either *constructive* in that it makes a copy of the stable capability or *destructive* in that it removes the stable capability from the caller's active directory.

COPY: creates a copy of a capability.

REMOVE: deletes a stable capability from an active directory.

DROP: deletes a transient capability from a c-list.

VIEW: These primitives bring a copy of a transient or stable capability, or an is-node into the address space of a process. Partial views are also facilitated. The purpose of these primitives is to allow a process to examine the 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.

ASSOCIATE: (de)associates a directory, manager or operation capability (from) with a cooperation class capability.

COMPARE: allows a process to compare two capabilities of the same type. It supports three comparison types: *compatibility check* by checking if the capabilities are linked/point to the same is-node and belong to the same cooperation class; *cooperation check* by checking whether the capabilities belong to the same cooperation class; and *kernel object check* by checking if the capabilities are linked/point to the same is-node.

MERGE: finds the union of the capcaps and the union of rights of two compatible capabilities and assigns them to the corresponding fields of the first capability. Two capabilities are compatible if they are linked/point to the same is-node and belong to the same cooperation class.

CHANGE-DIRECTORY: allows a process to change its active directory to either a new directory or to its initial/login directory.

As has previously been discussed, the manager definition and directory capabilities are slightly different from other capabilities. These capabilities contain pointers to manager definition and directory is-nodes, respectively. When one of these capabilities is created with the CREATE primitive, the is-node is created as well. These is-nodes exist in the system until either they are explicitly destroyed by using the DESTROY primitive, or all of the capabilities that point to them are deleted using the REMOVE or DROP primitives.

A transient capability comes into existence when a process copies or moves a capability from its active directory into its c-list (using a primitive 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. A process copies or 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 directory.

3.5 An Application: Distributed Guessing Game

This section discusses an application built on top of the prototype Gutenberg system. It is a distributed version of a guessing game in which the players have to guess a country using clues. The larger the number of clues used to guess a country, the smaller the score of the player. The game was designed so that it demonstrates most of the Gutenberg features, particularly the capability passing aspects of it and the use of the Interconnection Schema in controlling interprocess communication.

The players, the game controller and the clue givers are structured as separate objects managers. A player using hints, in the form of capabilities given to them by the game controller, attempts to establish ports to the clue givers in order to request the clue.

The game is organized as sessions. A session lasts from the point a player starts obtaining clues about a country and ends when the player guesses the country correctly or gives up for lack of further clues.

Each session of the game is viewed as a cooperative activity and as such is associated with a cooperation class capability S_i. The game controller keeps track of the cooperation class capability associated with the current session of a player. Clue givers use this cooperation class capability to give the clue appropriate for a particular session.

Each clue is provided by a separate clue giver. Each clue giver is associated with a unique cooperation class capability, and hence, is structured as a *class conservative* manager. The clue giver serves the operation *GetClue*. This operation is invoked by a player. *GetClue* is supported by a SendReceive port whose type allows the transferring of the cooperation class capability associated with a session as part of the request details.

A player is of a *creative* lifetype. For each player, a new manager process is instantiated. The capability to invoke the *StartGame* operation is registered in the players' initial active directory. This operation is served by the game controller. The *StartGame* operation is supported by a Receive port which is typed to allow the transferring of a

cooperation class capability and a SendReceive port capability.

Operation *GetGuess* is served by a player. It is invoked by the game controller to get the guess made by a player. *GetGuess* is done via a SendReceive port; a SendReceive port, and three capabilities, namely, an operation, a directory and a cooperation class capabilities, can be sent as part of the request details for *GetGuess*.

A player starts a session by creating a port `start-game-port` using the *StartGame* operation capability and invoking the RECEIVE primitive on the created port. The port is attached to the game controller. The game controller is instatiated, if it is not already active, to service the request. The game controller responds to the request by SENDing over `start-game-port` the cooperation class associated with the session S_i and the server-end port capability of a SendReceive port `get-guess-port` associated with the operation *GetGuess*.

By using SEND, the game controller permanently passes to the player the two capabilities. This does not carry any threat with respect to protection since the game controller creates a new cooperation class capability for each game session and keeps the client-end port capability of `get-guess-port` which allows the game controller to destroy the port at the end of the session. When it creates a new cooperation class for a session S_i, the game controller registers S_i in its active directory. The clue givers become aware of the new session by sharing the same active directory (*GameDir*) with the game controller (figure 6).

Subsequently, the game controller creates a SendReceive port, `get-clue1-port`, to the first clue giver, say *Clue1*, using the *GetClue* operation capability with the cooperation class capability *cl1* associated with the first clue. The game controller invokes *GetGuess* via `get-guess-port`. The port capability for `get-clue1-port`, the *GameDir/cl3* (*GameDir/cl3* is a directory capability for the directory *GameDir* and associated with the cooperation class capability *cl3*) and the cooperation class capability *cl2* are transferred as request details.

The player gets possesion of the passed capabilities by invoking GET-DETAILS on `get-guess-port`. The system stores the received capabilitities in the player's c-list. The player may try to use any of these capabilities to get the first clue. However, the player can get the clue only by using the port capability `get-clue1-port` and passing along S_i as request details. The directory capability *GameDir* is associated with a cooperation class capability identifying the third clue, *cl3*. Thus, it requires the player to possess *cl3* to change to the *GameDir/cl3*. Using the obtained clue, the player makes a guess

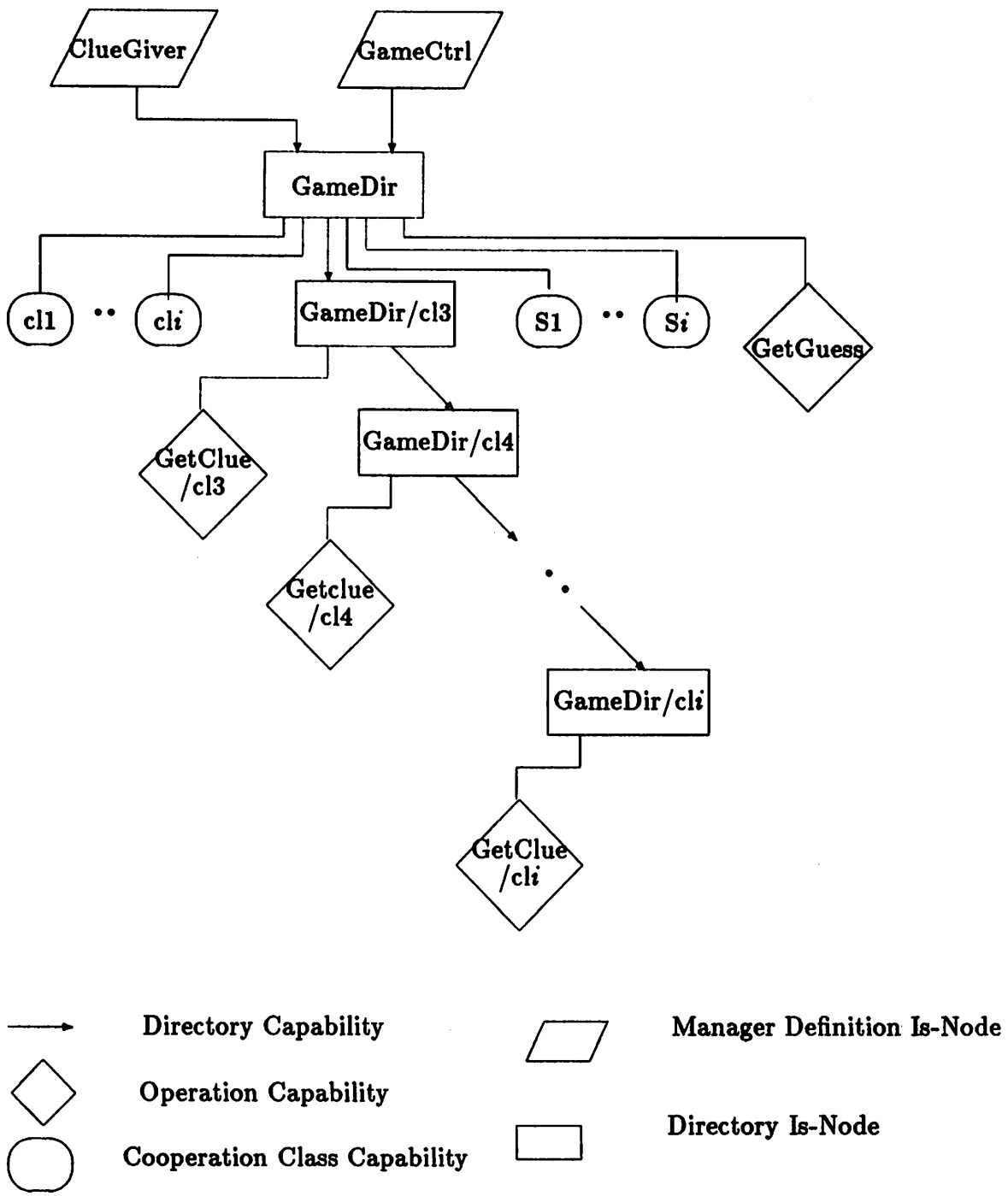

Figure 6: (Part of) Interconnection Schema for Distributed Guessing Game

and SENDs it to the game controller via **get-guess-port**. Since the three capabilities were received over a SendReceive port, they will be returned when the player responds with his guess. Hence, before the player sends its first guess to the game controller, it should register the directory and cooperation class capabilities in its active directory in order not to lose them.

If the guess is no good, then the game controller invokes *GetGuess* via **get-guess-port**. This time, it transfers the *GetClue* operation capability stored in its active directory, as request details. Since the operation capability *GetClue* is not associated with any cooperation class capability, it cannot be used without one to create a port to some clue giver. The player can use the previously received and registed cooperation class *cl2* for this purpose. In this way the player creates a SendReceive port to the second clue giver, say *Clue2*. Any attempt to use the directory capability will still fail. After receiving the second clue from *Clue2*, the player SENDs its new guess. Again, the lent operation capability is revoked and returned to the game controller.

If the guess is no good, the game controller invokes *GetGuess* again and passes the cooperation class capability *cl3* allowing the player to use the previously received *GameDir/cl3* directory capability to change its active directory to *GameDir*. In *GameDir*, the player finds the operation capability *GetClue* associated with *cl3* and a directory capability *GameDir* associated with *cl4*. Note that the previously received *GameDir/cl3* capability and the registered *GameDir/cl4* capability do not point to the same directory *GameDir*. It should be clear that the names of the capabilities are user-defined and are unique only within the contexts of an active directory or c-list.

Once again, the player can only use the *GetClue/cl3* operation capability to create a port to the third clue giver. If the player's guess is still no good, the game continues in the same manner. That is, the game controller passes more cooperation class capabilities allowing the player to traverse the Interconnection Schema visiting lower level directories and acquiring more *GetClue* operation capabilities to create ports to more clue givers. The game ends when either a player makes a correct guess or the game controller runs out of clues.

4 The Evolution of Gutenberg Primitives

In the experiment described in this paper, we succumbed to the inevitable temptation to constantly improve the system as it was developed. In spite of this, we have tried to

Port Type	Client Primitives	Server Primitives
any	CREATE-PORT REVOKE DESTROY	ACCEPT-REQUEST REFUSE
Send	SEND	RECEIVE
Receive	RECEIVE	SEND
SendReceive	SENDRECEIVE	GETDETAILS SEND

Figure 7: Initial port primitives used by clients and servers for each port type

distill the knowledge implied by our successive refinements, and to present it as clearly as possible.

In design and implementation-based experiments, each design decision is a hypothesis that the decision increases some measure of quality, though the metric is often hard to quantify. Furthermore, since there are many design decisions, it is very difficult to deduce causal relationships between the overall system properties and the decisions; i.e., it is difficult to validate the hypotheses of the experiment. Despite these difficulties it is important to experiment, to monitor design decisions as hypotheses, and to attempt to assess the contributions, both good and bad, of the decisions. In this section, we try to assign value to our decisions by tracking our improvements at stages and judging the decisions as bad decisions or failed hypotheses, and as good decisions or validated hypotheses.

Symmetry between Client and Server primitives improves Expressiveness

Very early in the development of the prototype kernel, we discovered that the initial set of port primitives (figure 7) allowed the client of the port to have more control than a server. We felt that the lack of *symmetry* in control between client and server was limiting the freedom of the server, and hence, the kind of supported applications. We hypothesized that symmetry between the client/server primitives would enhance the expressiveness of the system. One of the ways hypotheses are tested is to examine their implications and test them in order to validate the hypotheses. The implications of symmetry in this case was that there would be roughly an equal number of primitives and equal functionality for both client and server. Accordingly, we added three new primitives to the initial set of port primitives and enhanced the functionality of an

existing one. Two of the new primitives, EXAMINE and REJECT, and the enhanced one, the ACCEPT-REQUEST, are available to the server. The third new primitive, REVOKE, is available to the client. All three new primitives were easily implementable and the enhancement was a simple extension to existing code.

The symmetry hypothesis was corroborated by the fact that the new primitives allowed object managers to implement different scheduling strategies in servicing the operation requests received over ports. A server can use the EXAMINE primitive to decide which port to service next and which requests to refuse. A server of a port can permanently refuse to serve a port by invoking REJECT. REJECT is the dual of the DESTROY primitive available to the client of a port. Similarly, REVOKE is the dual of the server's REFUSE primitive.

A server gains access to a newly created port by invoking ACCEPT-REQUEST. Newly created ports are maintained in a queue until they become accessible to the server. ACCEPT-REQUEST was enhanced so that, instead of returning the next port on the queue, it allowed the selective acceptance of new ports based on the operation and/or the cooperation class for which they were created.

A server can make use of the enhanced ACCEPT-REQUEST primitive to delay the acceptance of ports associated with operation requests that currently cannot be satisfied. In this way, a server does not unnecessarily keep track of ports with currently unsatisfiable requests. Also, this improves performance because the search time for a port capability is reduced by having fewer port capabilities in the c-list. For the same reason, the response time of primitives that *view* port capabilities or examine the status of the accessible ports is decreased.

Uniform Represenation of Capabilities facilitates better Implementation

In the initial design, the cooperation class capabilities associated with a directory were maintained in the directory is-node, whereas those associated with an operation capability were part of the parameters of the capability. According to the uniformity hypothesis, the associated cooperation class capabilities should either become part of the parameters of the capabilities, or be maintained in the is-node to which the capabilities point. We chose the former. This choice immediately eliminated the need for the MODIFY-DIRECTORY-NODE primitive whose function was to add and delete the cooperation class capabilities associated with a directory.

Another effect of the uniformity hypothesis was the decision to group all capability types in a single structure for uniform manipulation. This decision motivated us to allow

the association of a manager capability with a cooperation class capability. This has the same semantics as the association of a cooperation class capability with directory capability. The functionality of the MODIFY-CAPABILITY primitives for the directory and manager definition capabilities was expanded to include the addition and deletion of associated cooperation class capabilities. However, this did not result in additional code complexity since the code existed already for the operation capability.

The validity of the uniformity hypothesis was further indicated by the introduction of the *s-component* (for *Schema component*). There is an s-component associated with each is-node. Each capability in the system referring to an is-node actually is linked/points to the s-component, not to the is-node itself. This allowed the implementation of a uniform timestamp-based scheme to uniquely identify the is-nodes in the system. The system-wide unique identifier of each is-node is stored in its corresponding s-component. The uniform system-wide unique identifiers simplified the construction and checkpointing of the Interconnection Schema. It also facilitated the distribution and partition of the Interconnection Schema in the completed design. An is-node exists always at its birth site. In the other sites an is-node is brought in from the birth site on demand. In any other site with capabilities pointing to this is-node, the node is represented by a corresponding s-component. Each site uses the s-components to store the necessary information to support concurrency control, garbage collection and is-node migration.

Simplicity of Representation Reduces Complexity of Implementation

The uniformity hypothesis had an impact on the manipulation of the associated cooperation class capabilities. Since a capability may have a number of associated cooperation class capabilities, we initially decided to maintain the list of associated cooperation class capabilities (*associated list*) as a linked list in the capability structure. This, as it turned out, had a number of serious drawbacks. First, the interface of a number of primitives was complex. In particular, the MODIFY primitive required an array of operation-capability pairs for modifying the association list. Second, the algorithmic complexity of a number of primitives was affected. Most of all, the MERGE and COMPARE primitives were complicated by the need to handle duplication of names in the association list when two capabilities were merged or compared. Finally, from a performance point of view, an additional search over the list of associated capabilities was required for almost all primitives. Copying a capability also became more expensive. This was an important consideration because copying a capability is required by a

number of schema and port primitives. Port primitives copy the transferred capabilites.

Thus, we felt that the use of many complicated structures was an indication of the lack of *representation simplicity.* We adopted the hypothesis that representation simplicity (without loss in semantics and flexibility) that minimized the need and the effort to transform one representation to another would reduce the complexity of algorithms underlying the primitives. The simplest representation has a *linear* structure implementable using sequential storage. It should be noted that linear structures, as opposed to non-linear, complex ones, can be passed over a communication channel (a mailbox in our prototype) without any transformation. One way to restore the linear structure of a capability was to allow a fixed number of capability associations, in which case the association list could be represented by an array in the capability structure. However, this alternative is static, it unnecessarily increases the size of a capability structure while wasting space, and above all, it does not solve the performance problems or the complexity problems. A second alternative was to allow a capability to be associated with at most one cooperation class capability. The effect of having a capability associated with a number of cooperation class capabilities can be obtained by having the same number of independent single capabilities associations.

We adopted the second alternative to test the extreme case of the representation simplicity hypothesis. This decision solved all four problems above by simplifying the system conceptually while leading to a simplification in the implementation. The naming of the capabilities was improved. Capabilities in the system are uniquely specified by the combination of the user-defined local name of the kernel object they protect, the user-defined local name of the cooperation class to which they belong, and their type. By using a structure to group all three components of a capability specification, the interface of most primitives was simplified, requiring, on average, four formal parameters including the *status.* The coding complexity of all the primitives was also reduced. Furthermore, code duplication was eliminated by implementing only the generic primitives to handle all capability types.

Primitives with Simple Functionality lead to Better Implementation

Since representation simplicity caused simplification both in the primitives, which enhances the programmability of the system, and in coding, which eases the debugging and testing of the system, we expected that *functional simplicity* would cause even further simplifications. Our hypothesis led to the splitting of two primitives, one port and one schema primitive, into four new primitives.

The ACCEPT-REQUEST primitive was split into the ACCEPT-REQUEST and QUERY-PORTS. The initial ACCEPT-REQUEST primitive was overloaded, exhibiting two different behaviors depending on the number of input parameters. The schema primitive MODIFY was split into the two primitives MODIFY and ASSOCIATE. The new MODIFY primitive just renames and changes the parameters of a capability. The ASSOCIATE primitive associates (dissociates) a capability with (from) a cooperation class capability.

Furthermore, based on the functional simplicity hypothesis, we decided to drop complex and rarely used primitives whose functionality could be obtained by combining other simpler primitives. This decision caused the elimination of the MODIFY-MANAGER-DEFINITION-NODE. Modifying a manager definition, particularly while it is active, is a very complex and expensive operation because it requires back pointers to all operation capabilities in order to update the specification of the operation in all the affected operation capabilities. The same functionality could be obtained by combining the two existing primitives, CREATE-MANAGER and DESTROY-MANAGER at the cost of creating and redistributing the operation capabilities. However, the cost of creating and redistributing the operation capabilities under the assumption that modification of a manager definition is rare, is very small compared to the cost of maintaining back pointers. Of course, the problem of redistribution is left to the user. It should be pointed out that optimizations, possible in the case of a single primitive, might not be possible in the case of two primitives which provide the same functionality.

Oversimplification of Primitives can affect Performance

Functional simplicity may produce a large number of simple primitives resulting in the possibility that some simple effects required more than one primitive. For example in our initial version, the moving of a capability from an active directory to a c-list required the invocation of the HOLD primitive to create a transient copy of the stable capability followed by the invocation of the REMOVE primitive to remove and destroy the stable capability from the active directory. Similarly, registering while keeping a transient capability, required the invocation of the REGISTER and HOLD primitives.

These indicated that *oversimplification* may be a threat to performance. We discovered that our hypothesis was valid during the implementation of the mailbox facility which supported the primitive invocation mechanism. The cost of invoking a primitive included one memory allocation and two copy operations to and from the mailbox by both the kernel and the invoking object manager. Of course, different implementa-

tions entail different costs for invoking a primitive. However, the recognition of this cost motivated us to provide constructive and destructive versions of some primitives. Specifically, a destructive HOLD was equivalent to a HOLD followed by a REMOVE in the previous version and a constructive REGISTER was equivalent to a REGISTER followed by a HOLD. This was done after making sure that none of the other hypotheses was invalidated. Thus, the additional functionality did not result in a complex interface or complex implementation which would defeat the purpose of eliminating the extra primitive invocation.

5 A Critique of the Gutenberg System

In this section, we recap the knowledge gained from the experiment with respect to the two initial goals, first, to evaluate the expressiveness and programmability of Gutenberg, and second, to discover of difficulties in implementing its logical level. Based on the acquired knowledge, we attempt to predict the performance of the distributed Gutenberg.

5.1 Expressiveness and Programmability

The prototype was debugged and tested by implementing a simple menu-driven message exchange facility which allowed for different invocation scenarios and transferring of schema and port capabilities. The development of this facility proceeded in parallel with the development of the first version of the system.

Several applications of various complexity were built for testing and demonstration purposes. One of the first non-trivial applications was the distributed guessing game presented in section 3.5. Other applications include a file system which uses ObServer [Skarra87] as its object store, and a data-driven database query execution over a relational database. In the latter, each relation in the database is maintained by a different object manager. Each node of the query evaluation tree is implemented as a separate object manager.

All three applications exhibit different kinds of interactions and have different communication topologies. For example, cooperation class capabilities in the game are used to identify the clues and their clue givers, and as passwords to navigate the Interconnection Schema; the file system uses the cooperation capabilities to represent file objects; and the database query uses cooperation class capabilities to identify the

relations and to correctly establish the query evaluation tree.

We chose to build these applications on top of Gutenberg because they exhibit the basic interactions found in a number of classes of applications. For example, the interactions involved in the game are similar to those found in Computer Aided Instruction Systems. In fact the game itself can be viewed as a computer tutor. The interactions found in the message exchange facility, file system and database query are typical of system software. The combination of the game and the data-driven database query execution defines the basic set of interactions found in structuring Office Automation facilities such as a Calendar [Chrys88]. Experience with these applications led us to believe that the final set of primitives facilitates the applications Gutenberg was targeted to support, i.e., systems made up of many distributed cooperating modules which execute asynchronously and concurrently.

The fact that the kernel was not distributed at the time that these applications were written, has no effect on the applications, and hence, it does not minimize the evaluation of the *expressiveness* of the kernel. All applications built can be distributed without any modification since each object manager can potentially execute anywhere in the distributed environment. However, as we learned from the experiment, the performance effects of some details of the distribution are impossible to predict through the logical design of the system and by building paper applications. Thus, a complete evaluation of the system has to be delayed until the system is fully distributed.

The game and the file system were designed and fully implemented within two weeks by two people, one of whom had not worked before on the Gutenberg implementation. The functional addressing for establishing ports significantly simplifies the organization of an object manager compared to other systems, for example, that of a process in UNIX using sockets. The only inconvenience in building an application in Gutenberg and using ports is that complex data structures cannot be passed over a port and the programmer has to convert them to and from raw bytes and vice versa. But this is also true of other systems which support port-based communication.

In terms of programmability, the current structure of the Gutenberg object manager needs improvement. Using processes with a single thread without software traps is not convenient for dealing with asynchronous port operations. Minimally, software traps (as in UNIX) associated with port handling are necessary.

5.2 Algorithmic and Code Complexity

In terms of coding complexity, the code that implements the kernel structures unique to Gutenberg, namely, the Interconnection Schema and typed ports is relatively simple and straightforward. The few lines of cryptic code are in the functions which handle response to asynchronous requests occuring while the requesting manager process is blocked on another request.

In the beginning, we expected that the schema primitives which update the Interconnection Schema would be more expensive than the port primitives. This was supported by the facts that garbage collection entailed some overhead in maintaining the reference counters. Also, destruction of an is-node might cause other is-nodes to be destroyed resulting in cascading destruction of more is-nodes. However, we discovered that the update of the Interconnection Schema was only as expensive as sending a message containing a single capability. However, transferring a number of capabilities, particular exclusive ones, was not as inexpensive as would appear.

The transferring of capabilities does not affect the Interconnection Schema with respect to its contents but the construction of a message containing capabilities, requires updating reference counters. The construction of each capability involves a search, a validation, and the creation of a new capability (i.e., memory allocation, copy, and update of reference counters). This cost should be about the same for all capability-based systems which support transferring of capabilities. However, there is an additional cost in Gutenberg due to the need to keep track of the temporary transfer of capabilities and the permanent transfer of exclusive capabilities until their reception. The need to trace the lent capabilities is obvious. The need to keep track of exclusive capabilities until their delivery is not so obvious. Exclusive capabilities, as opposed to non-exclusive ones, are removed from the sender's c-list. In order to avoid name conflicts in the event that these capabilities are returned to the sender, they are considered to be in the c-list (until they are delivered) but they cannot be used. Name conflicts are possible because of the asynchronous communication.

Perhaps the most complex algorithm in the kernel relates to the temporary transfer of capabilities. The source of the complexity is the revocation and return of the port capabilities and in particular, of the SendReceive ports. The revoking and returning of a temporarily transferred (*lent*) capability might cause other capabilities to be revoked. The reason for this is that in order for a port capability to be returned, the corresponding port must not have any pending requests and outstanding capabilities. Otherwise, they

too need to be revoked and returned. The kernel guarantees the proper return of the lent port capabilities by delaying their revocation until they are in a state appropriate for revocation. Furthermore, a port capability may be transferred over a number of other ports. The kernel keeps track of these transfers by maintaining a stack of all the ports over which the lent capability has crossed. Since both server and client end capabilities can be lent, there is a pair of such stacks for each port.

5.3 Predicted Performance of Distributed Gutenberg

Although we have not yet completed the distribution of the prototype, we believe that we are in a position to predict the overhead that would be incurred based on the structure of the prototype system. Since the kernel and object managers do not share address space, the kernel/object manager interactions in the prototype are similar to the kernel/kernel interactions in the distributed system. In a distributed system, an instance of the Gutenberg kernel is running on each site in the system (figure 8). That is, the implemented primitive invocation mechanism in a single node is practically the same as the one being implemented for remote invocations. For example, the VIEW primitive, which brings a copy of a capability or an is-node into the address space of an object manager, uses the mailbox facility, and thus, it needs to linearize, i.e., to convert to a sequence of bytes, the viewed kernel object. Similarly, the kernel module which brings a copy of a capability or is-node into another kernel uses the communication network facility, and thus, it also needs to linearize the transferred kernel object.

The bottleneck in Gutenberg is expected to be the update of the Interconnection Schema which is a logically unified but physically distributed structure. The distribution of the Interconnection Schema is based on the primary copy scheme [Bernstein87]. An is-node always exists at its birth site. Manager definition is-nodes are replicated only at the sites in which manager processes from these can possibly be instantiated. Directory is-nodes are brought into a site from the birth site on demand. The kernel at the birth site of an is-node is responsible for managing the is-node.

Based on our experience so far, the Interconnection Schema is rarely updated, in comparison to the port operations. Furthermore, the transfer of capabilities is not expected to be as frequent as the rest of the communication. These lead us to believe that the overheads incurred by the distribution would not negate the benefits of the port-based protection in which the request for an operation on a user-defined object is validated at the site where the request originated and not at the site where the object

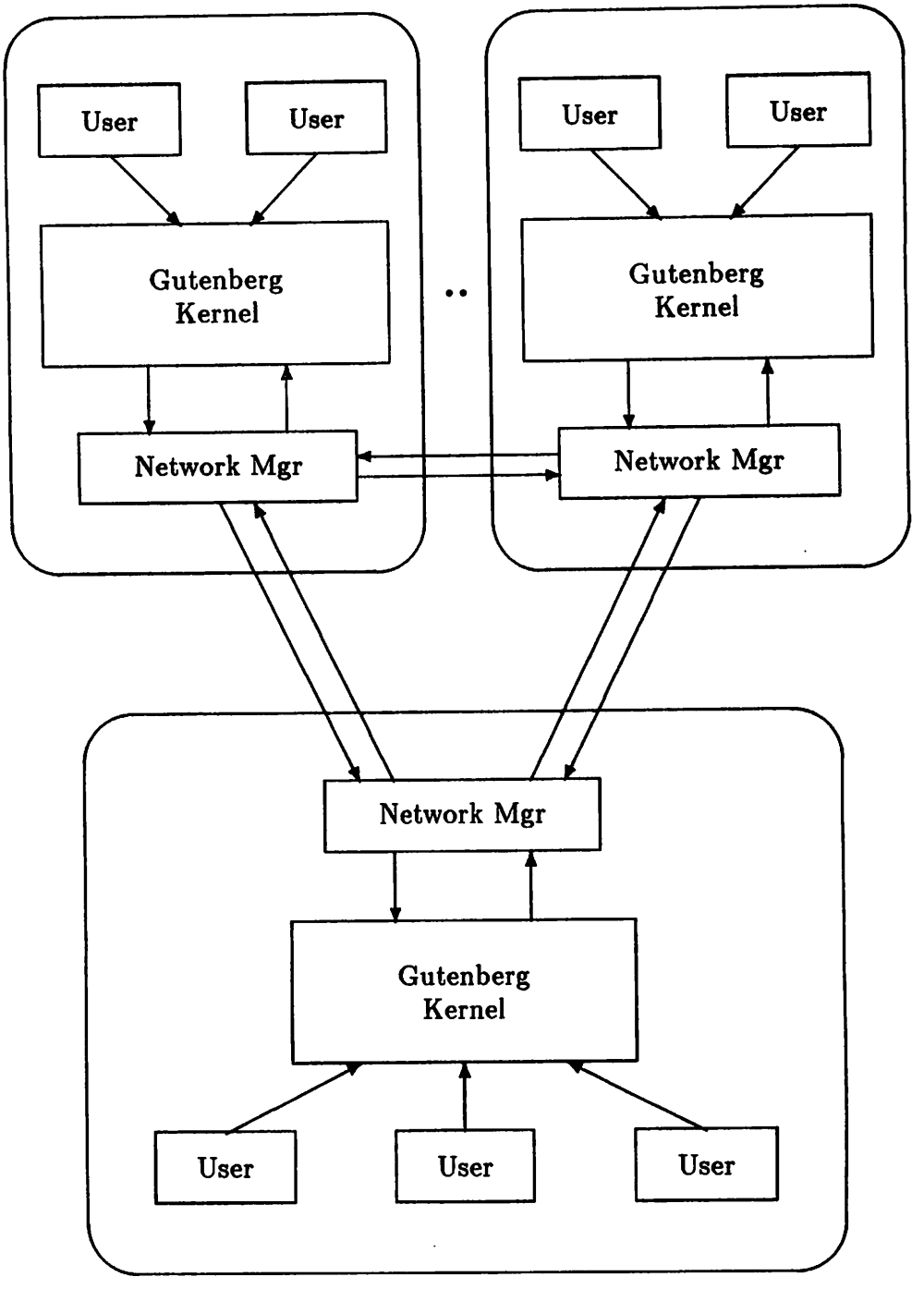

Figure 8: Distributed Gutenberg System Organization

resides.

Based on this observations, we predict that the positive experience with the current version of Gutenberg will carry over to the fully distributed version as well.

6 A COMPARISON WITH RELATED SYSTEMS

A large part of the improvement in programming languages has come from the promotion of data abstraction and procedural abstraction as major tools for structuring modules. This led to the adoption of abstraction and encapsulation mechanisms in Gutenberg, as in other systems, e.g. Argus [Liskov82], NIL [Strom83]. In these two systems, the mechanism for dynamic module interconnection control is built within a programming language. However, the approach taken in Gutenberg separates process interconnection control from programming languages. In this respect, Gutenberg differs with the Conic environment [Magee87] which combines language-based and operating system approaches for dynamic module configuration. 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 [Wulf74] and iMAX [Cox81], in that it adopts a non-uniform object-orientation: Communication between object managers is object-oriented but intramanager communication may not be object-oriented.

A few systems provide port-based communication facilities using functional addressing [Stemple86], but none ties access control so closely to communication as Gutenberg does. The Accent system is port-based and supports asynchronous communication with process transparency [Rashid81]. Communication in Gutenberg is also similar to the mechanisms used in Intel iAPX-432 [Cox81], DEMOS [Baskett77] and NIL [Strom83]. In all these systems, apart from NIL, even though a communication link could be typed, thus restricting its use, the concept of restricting access to shared objects by restricting the creation of communication channels is not supported directly, as in Gutenberg. We believe that prior to the design of the Mercury system [Liskov88], Gutenberg was the only system which combined message passing and RPC semantics in a clean and uniform way, thus allowing synchronous and asynchronous RPC semantics.

Gutenberg and Clouds [Spafford84] differ in almost all respects. For example, Clouds objects are passive and accessed through operation invocations made by processes. No kernel supported communication facility is provided. Although, Gutenberg

has similarities with Eden [Almes85], and MACH [Baron84], the main difference comes from the provision of the Interconnection Schema in Gutenberg. As mentioned earlier, a unique feature of Gutenberg is the Interconnection Schema, which contains stable capabilities in a unified structure controlled by the kernel. Other systems, such as Hydra, iAPX 432, CAL [Lampson76], and Amoeba [Tanenbaum86] 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 [Lampson76]. 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.

7 Conclusion

Influenced by the goals of the implementation and in order to accelerate the development of the Gutenberg prototype and improve its presentation, we decided to build the kernel on top of an existing kernel and make use of other available tools/facilities in the environment. This decision led us to underestimate the time and the number of people needed to implement the system. We spent much time figuring out some of the tools in order to make them work properly. In addition, we ended up making decisions about the prototype implementation based on the limitations of the underlying operating system and other facilities rather than based on the functionality of Gutenberg. For example, we had to design and implement the mailbox facility for a second time after discovering that the Ultrix system does not allow more than six shared segments to be attached to a single process.

The choice of the representation of the kernel objects played an important role in the development of the system. The appropriate representation simplified the interface and the implementation of the logical level of the system without a loss of functionality. Furthermore, it minimized the amount of effort needed to transform the representation to and from a linear form which is required when a kernel object is moved from one node to another, from one process to another, or from memory to disk. Most of the overhead in the system is due to the transformation from one representation to another.

During the experiment, the Gutenberg primitives evolved in two ways. The initial set of primitives was expanded in terms of numbers and changed in terms of the semantics of the primitives.

We believe that, so far, we have achieved both goals of the Gutenberg experiment, that is to evaluate the expressiveness and programmability of the Gutenberg kernel, and to discover difficulties in implementing the Interconnection Schema and typed ports.

Experimentation with complex systems is difficult in general, and complex programs are no exception. Operating systems are especially difficult because of their general purpose nature and the multitude of disparate services they provide, from memory and file management to system structuring mechanisms. The design and implementation of complex systems, including operating systems, can be cast as experimentation, but can rarely achieve the clarity of experiments in natural science.

In the experiment described in this paper, we succumbed to the inevitable temptation to constantly improve the system as it was developed. In spite of this, we have tried to distill the knowledge implied by our successive refinements, and to present it as clearly as possible. While our analysis may still suffer from loose metrics and a certain amount of subjectivity, we hope it contributes to a better knowledge of the principles of distributed system building.

8 Acknowledgements

The Gutenberg experiment has benefited from a number of contributions. Steve Vinter deserves special mention since the initial design of the Gutenberg Kernel formed his Ph.D. thesis. David Briggs contributed ideas during the first phase of the project. Hanuma Kodavalla actively participated in the development of both the first and second version of the Gutenberg Prototype. V.R Govindarajan implemented the checkpointing module of the Kernel. Zhimin Shi is currently working on the third version of the prototype.

9 References

[Almes85] G. Almes, A. Black, E. Lozowska, J. Noe, 'The Eden System: A Technical Review,' *IEEE Transactions on Software Engineering*, vol. SE-11, no. 1, January 1985.

[Baron85] Baron,R., Rashid, R., Siegel, E., Tevanian, A., Young, M.W., 'MACH-1: A Multiprocessor-Oriented Operating System and Environment', CMU Technical Report, 1985.

[Baskett77] F. Baskett, J. Howard, J. Montague, 'Task Communication in DEMOS,' *Proceedings of the 6th ACM Symposium on Operating System Principles*, November, 1977.

[Bernstein87] Bernstein, P., Hadzilakos, V., and Goodman. N., 'Concurrency Control and Recovery in Database Systems,' *Addison-Wesley Edition*, 1987.

[Chrys86] P. K. Chrysanthis, K. Ramamritham, D. W. Stemple, S. T. Vinter, 'The Gutenberg Operating System Kernel,' *Proceedings of the First ACM/IEEE Fall Join Computer Conference*, November, 1986.

[Chrys87] P. K. Chrysanthis, 'The Gutenberg Prototype Operating System (version 1.4),' *Gutenberg Project Report*, Computer and Information Science Department, University of Massachusetts, September 1987.

[Chrys88] P. K. Chrysanthis, H. Kodavalla, K. Ramamritham and D. W. Stemple, 'The Gutenberg Prototype Operating System (version 2.0),' *COINS Technical Report TR-88-25*, University of Massachusetts, March 1988.

[Chrys88] P. K. Chrysanthis, D. W. Stemple, K. Ramamritham ' Structuring Office Systems Using Gutenberg' *COINS Technical Report*, University of Massachusetts, April 1989.

[Cox81] G. Cox, W. Corwin, K. Lai, F. Pollack, 'A Unified Model and Implementation for Interprocess Communication in a Multiprocessor Environment,' Intel Corporation, 1981.

[Lampson76] B. W. Lampson,H. E. Sturgis, 'Reflections on an Operating System Design,' *Communications of the ACM*, vol. 19, no. 5, May, 1976.

[Liskov82] B. Liskov, R. Scheifler, 'Guardians and Actions: Linguistic Support for Robust, Distributed Programs,' *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages*, January, 1982.

[Liskov88] B. Liskov, T. Bloom, D, Gifford, R. Scheifler, W. Weil, ' Communication in the Mercury System,' *21st Annual Hawaii Conference on System Science,* January 1988.

[Magee87] J. Magee, J. Kramer, M. Sloman, 'Constructing Distributed Systems in Conic,' *(to appear in) IEEE Transactions on Software Engineering* (Imperial College Research Report DOC97/4, October 1987).

[Rama83] K. Ramamritham, S. T. Vinter, D. W. Stemple, 'Primitives for Accessing Protected Objects,' *Proceedings of the Third Symposium on Reliability in Distributed Software and Database Systems,* October 1983.

[Rama85] K. Ramamritham, D. W. Stemple, S. T. Vinter, 'Decentralized Access Control in a Distributed System,' *Proceedings of the 5th International conference on Distributed Computing Systems*, May 1985.

[Rama86] K. Ramamritham, D. Briggs, D. W. Stemple, S. T. Vinter, 'Privilege Transfer and Revocation in a Port-Based System,' *IEEE Transactions on Software Engineering*, vol. SE-12, no. 5, May 1986.

[Rashid81] R. Rashid, G. Robertson, 'Accent: A Communication Oriented Network Operating System Kernel,' Carnegie-Mellon University Technical Report, April, 1981.

[Skarra87] A. Skarra, S. B. Zdonik, S. P. Reiss, 'ObServer Documentation - Base System', *Technical Report*, Brown University, January 1987.

[Spafford84] E. Spafford, M. McKendry, 'Kernel Structures for Clouds,' *TR-GIT-ICS-84/09*, School of Information and Computer Science, Georgia Institute of Technology, March 1984.

[Stemple83] D. W. Stemple, K. Ramamritham, S. T. Vinter, T. Sheard, 'Operating System Support for Abstract Database Types,' *Proceedings of the 2nd International Conference on Databases*, September, 1983.

[Stemple86] D. W. Stemple, Vinter, S., K. Ramamritham, 'Functional Addressing in Gutenberg: Interprocess Communication Without Process Identifiers,' *IEEE Transactions on Software Engineering*, vol. SE-12, no. 11, December 1986.

[Strom83] R. Strom, S. Yemini, 'NIL: An Integrated Language and System for distributed Programming,' *Proceedings of SIGPLAN '83, Symposium on Programming Languages*, August, 1983.

[Tanenbaum86] A.S. Tanenbaum, S.J. Mullender, R. Renesse, 'Using Sparse Capabilities in a Distributed Operating System,' *Proceedings of the fifth International Conference on Distributed Computing Systems*, May 1986.

[Vinter85] S. T. Vinter, 'A Protection Oriented Distributed Kernel,' Ph.D. Thesis, University of Massachusetts, August 1985.

[Vinter86] S. T. Vinter, K. Ramamritham, D. W. Stemple, 'Recoverable Communicating Actions,' *Proceedings of the fifth International Conference on Distributed Computing Systems*, May 1986.

[Wulf74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, F. Pollack, 'HYDRA: The Kernel of a Multiprocessor Operating System,' *Communications of the ACM*, vol. 17, no. 6, June 1974.