1988

Supporting Objects in a Conventional Operating System

Prasun Dewan

Report Number:
88-762

SUPPORTING OBJECTS IN A CONVENTIONAL OPERATING SYSTEM

Prasun Dewan

CSD TR-762
April 1988

Supporting Objects in a Conventional Operating System

Prasun Dewan
Department of Computer Science
Purdue University
W. Lafayette, IN 47907

ABSTRACT

A simple approach is presented for introducing objects in a conventional operating system. Objects are created as combinations of conventional processes and files. Like processes, they are active agents capable of executing code and communicating with other objects. Like files, they are persistent, have a protected name in the file system, and can be opened and closed for access. Motivation for supporting objects in a conventional system is presented. The basic elements of our approach are described together with the rationale for our decisions. An implementation on Unix[1] is discussed. Object-based programming at the system level is contrasted with object-based programming at the language level. Comparisons are made between our approach and related work.

1. Introduction

Object-based programming languages have demonstrated two approaches for supporting objects: The first, illustrated by Smalltalk [8], is to create a new language based entirely on the concept of objects, while the second, exemplified by C++ [14], and Objective-C [4], is to extend a conventional language with features to support object-based programming. The first approach offers the potential for a more uniform language, uncompromised by the constraints of integrating objects with existing facilities in the base language. On the other hand the second approach lets a programmer incrementally test object-based ideas without sacrificing familiar data and control structures of conventional languages. Moreover,

[1]Unix is a registered trademark of AT&T Bell Laboratories

software written in the new language can be combined with the software written in the base language. Finally, the exercise of integrating object-based programming with conventional programming has lead to useful hybrid concepts such as strongly-typed object-based languages [15].

Object-based operating systems (Clouds [13], Hydra [16], and Eden [1]), on the other hand, have only tried the first approach of creating new systems based entirely on the concept of objects. These systems offer unconventional methods for supporting operating system components such as processes, files, and access control. Moreover, they are mainly prototype kernels instead of full systems, and do not offer alternatives to the large number of facilities offered by existing systems. As a result, they have not been used to the extent needed to demonstrate the usefulness of supporting objects.

The idea of integrating objects in conventional operating systems is, therefore, attractive. It can let the programmer explore the use of objects without the fear of sacrificing time-tested components of conventional systems. Moreover, as in programming languages, it may lead to useful hybrid concepts combining features of object-based and conventional operating systems.

In this paper we present a simple approach for introducing objects in a conventional system. First we discuss the motivation for supporting objects in a conventional system. We then describe the main concepts of our approach and discuss the rationale for some of our choices. Next we outline an implementation of a subset of the approach on Unix. Finally, we discuss various aspects of our approach, compare it with related work, and discuss potential directions for future research.

2. Motivation

In a conventional operating system, files, and the processes that manipulate them, are kept separate. This separation leads to at least three problems:

- Files are untyped, that is, there is no information in a file about the processes that can manipulate them. Thus any process that has access to a file may manipulate it. As an example, consider an appointment file that stores a list of appointments of a user. Only processes executing programs written specifically to manipulate such a file should be allowed to access it. However such a file may also be accessed by compilers, text formatters, and other processes with appropriate access

rights.

- Each process that correctly manipulates a file needs to know the syntactic and semantic constraints of the data stored in the file. This information is hard coded in the program executed by the processes. As a result, it is hard to change the syntax and semantics of data stored in files. Continuing with the appointments example, each program that manipulates an appointment file needs to detect syntax errors such as a missing appointment time, and semantic errors such as an appointment time that has elapsed. If the semantics of appointment lists were changed to, say, require that a change to an appointment in one file is reflected in the appointment files of all other users involved in the meeting, then all programs that manipulate appointment files need to be located and changed to incorporate the change.
- At any time, a copy of the information in a file may also be loaded in the data structures of a process. These copies may be manipulated independently, and can, therefore, become inconsistent. Moreover, attempts to keep these copies consistent may lead to wastage of computer resources, since the process needs to poll the file for changes made by other processes. For instance, a process that displays a user's appointements for the current day must read the appointments file periodically to ensure that the information it displays is consistent with the current contents of the appointments file. Reading the file frequently would consume computer resources while reading it infrequently can cause a user to miss an appointment.

Operating systems such as Demos [2], Charlotte [7], Berkeley Unix, and Mach [9] that support the client-server paradigm of interaction have illustrated the use of object-based programming to solve some of these problems. Information that would otherwise be kept in a file may instead be encapsulated in the data structures of a *server* process, which responds to messages from clients wishing to manipulate the information. The server is solely responsible for managing the information and defining its syntax and semantics.

Returning to the appointments example, an appointment server may be created for each list of appointments. Processes wishing to manipulate such a list send messages to the server, which makes the requested changes according to the syntax and semantics of the list. An example of such a message may

be the `add_appointment` message, which a client may send to add an appointment to the list. A server may process this message by checking the new entry for errors, such as a missing or invalid appointment time, adding the entry in case of no errors, and sending messages to appointment servers of other users involved in the meeting. The definition of the syntax and semantics of appointment lists is coded in the program that the appointment servers execute, and may be changed by modifying only this program.

The client-server model of process interaction, however, is not by itself sufficient for encapsulating all information:

- Servers are temporary process and thus unsuitable to encapsulate persistent information. A persistent server can be simulated by temporary *incarnations* of the server. Each incarnation is a new process that executes the server's program and checkpoints its persistent state in a data file, which the next incarnation can read to initialize its state. A problem with this scheme is that any process with appropriate access rights can modify the data file. Moreover, errors can be made in connecting an incarnation to its data file.
- Access to servers is not protected from arbitrary clients. Most server-based systems do require that a client have a capability to a port advertised by a server before it can send a message to the port. However, the operating system distributes these capabilities to clients without distinguishing between different classes of clients. For instance, Berkeley Unix allows any process to connect to a socket on which a server is listening. Similarly, a Demos or Charlotte name server gives a link registered by a server to any client that requests it. As a result, secure information in these systems needs to be accessed via protected files.
- An incarnation of a server needs to be always active since requests from clients may arrive at random times. However, only a limited number of processes can be active at any one time. Therefore, only a small amount of information can be accessed through servers. The majority of information needs to be accessed via files.

These problems can be solved by supporting protected, permanent servers that can be selectively activated and passivated. The next section presents an approach for supporting such servers in conventional operating systems.

3. Approach

Model of a Conventional System

Before we describe our approach, we need to define a conventional operating system. We use this term to refer to an operating system that supports the following:

- a set of resource-independent file operations (that is, applicable to file and non-file resources) such as `file_create`, `open`, `close`, `rename`, and `delete`,
- access rights that include the `read`, `write`, and `execute` rights,
- access control based on access-lists,
- a set of process operations such as `process_create`, `suspend`, `resume`, and `kill`,
- IPC facilities allowing processes to send and receive messages from *ports*, a general term we use for Unix sockets, Demos and Charlotte links, Mach ports, and other mailboxes defined by operating systems.

Objects

Conventional systems define a general file interface that can be used to access most resources such as files, directories, and devices. Resources accessed through this interface have several useful properties such as user-defined names, persistence, and protection. However, these systems treat processes as second-class citizens accessed through a special interface that does not support these properties, the very properties that servers need to encapsulate general information. Our approach gives processes these properties by making them first-class citizens accessed through the file interface. In the rest of this discussion, we shall refer to these upgraded processes as objects and conventional processes as simply processes.

An object possesses all properties of a process. Thus it executes some program, and can use all facilities available to a process. In particular, it can use the available IPC facilities to communicate with other processes and objects. However, unlike a process, but like a file, an object has a protected name in the file system, is persistent, and is associated with both an active and a passive state.

Activation and Passivation of Objects

When an object is passive, its persistent data is saved in an internal *data file* maintained by the operating system for the object. A data file is accessed as a regular file, but does not have an external name, and thus cannot be opened by arbitrary processes. An object may however open its data file by making a special `open_data_file` call.

Before a process or object starts communication with an object, it needs to `open` the object for access. This call activates the object if it is passive, increments the reference count of the active object, and returns a file descriptor that refers to the object. When the client no longer needs to access the object, it calls `close`, which decrements the reference count of the object.

An active object cannot be automatically passivated when its reference count goes to zero, since it may be in the middle of uninterruptible conversations with other objects, processes, or the user. In general, the system does not know which conversations of an object are uninterruptible. Therefore, when the reference count of an object goes to zero, the system sends the `ref_cnt_zero` message the object to signal this event. The object may postpone passivation if its current state does not allow it. Later, when its state does allow passivation, it may make the `passivate` call to ask the system to passivate itself. Before an object makes this call, it needs to establish its *passive invariant*. For instance, a Unix object needs to close all sockets it is listening on, a Demos or Charlotte object needs to deregister its services from a name server, and an interactive object needs to close all windows displayed by it.

Between the time an object receives the `ref_cnt_zero` message and it makes the `passivate` call, some client may try to open it. The system blocks the client until the object calls passivate, when it simply reactivates the object and unblocks the waiting client. A drawback of this scheme is that the client has to wait until the object reaches a state that allows passivation. An alternative

approach would be to ask the object to ignore the `ref_cnt_zero` message and unblock the waiting client. However, this approach makes programming of the object hard, since the object may be in the middle of establishing its passive invariant when it is reopened. We chose the first approach because we expect such situations to occur rarely.

An active object does not start executing at the point it was passivated, since it may need to establish an *active invariant* before it can continue its activity. For instance, a Unix object may need to create sockets for communication with other objects and a Charlotte object may need to register its links with a name server. Therefore, when an object is activated or created it needs to start execution at the beginning of the code that establishes its active invariant.

We assume that all objects that passivate themselves are servers that execute a main body of the following form:

```
begin /* main */
      establish_active_invariant;
      loop
            wait for message or user input;
            service event
      end
end /* main */
```

Therefore, when an object is activated or created, it starts execution of the main body of its program.

A client that has activated an object may need to know if the the object has established its active invariant. For instance, a Unix process interested in communicating with the object may need to know if the object is listening on its sockets. Therefore, we also support the synchronous `open_wait` call that waits until the newly activated object executes the `ready` call to inform the system that it has established its active invariant.

Systems that support persistent objects use one of two approaches for saving and restoring the state of the object. The first, used by Eden, makes an object responsible for saving and restoring its persistent state, while the second, used by Smalltalk and Argus [12], places this responsibility with the system. We adopt the first approach, for two reasons:

- An object can save only that part of the state that is persistent. An operating system, on the other hand, would have to save the complete state of the object since it does not know which data structures are permanent. This problem is solved in Argus by requiring that the programmer explicitly declare certain data structures as persistent. We cannot assume such language support since we allow objects to be coded in any programming language supported by the system.
- The alternative approach does not support evolution of objects when the programs they execute are upgraded. When a program that an object executes changes, the offsets, types, and names of the saved variables may change. In general, an operating system responsible for restoring the state would not know how to map the saved state to the new state. (It could handle the special case of changes to offsets by using, as in Smalltalk, compiler help mapping old offsets to new offsets). Our approach, on the other hand, allows evolution of objects by making the new program responsible for mapping the saved persistent state to the new data structures.

A potential disadvantage of our approach is that the the object has to do extra work invoked in saving and restoring its permanent state. This work is trivial for the part of the permanent state comprising of statically allocated data structures. It is more complicated when the permanent state consists of references to dynamically allocated data structures, which need to be recreated whenever the object is activated.

Object Creation

An object is created by a special *object_create* call that is a cross between the *process_create* and *file_create* calls. It takes all arguments of the former, such as the name and arguments of the program to be executed. It also takes arguments of the latter, such as the file name and access list of the new object. Thus an appointments server may be created by a call of the form

```
object_create ("/usr/joe/appts", 700, "/usr/bin/Appts", "-1")
```

where `/usr/joe/appts` is the name of the object, `700` is a description of the access list associated with the object, `/usr/bin/Appts` the name of the program that the object executes, and `-1` is an argument to the program. A newly created object starts in the active state.

Object Activation in a Distributed System

A distributed system supporting a network file system needs to decide which machine an object should be activated on. We considered several choices:

- The machine on which the client that activates it resides. This approach guarantees that at least one process communicating with the object is on the same machine as the object. However, it requires that the object's program be executable on each machine on which it is opened. Moreover, it does not accommodate programs using machine-specific file names since they may be executed on more than one machine.
- The machine on which the object name resides. Thus object `/usr/joe/appts` would always execute on the machine on which the directory `/usr/joe` resides. It would be the owner's responsibility to ensure that the object's program is executable on this machine. This approach is consistent with file activation schemes that activate files on machine on which their names reside. Moreover, it lets a user migrate objects by renaming them. (Migration of active objects may be disallowed if the underlying system does not support process migration). However, it is possible for an object can be activated on a machine on which none of its clients reside. This situation would occur frequently when objects are created and opened from workstations that cannot store object names. Moreover, since objects may migrate, this approach also does not allow the use of machine-specific names.
- The least loaded machine on which the program can execute. This scheme requires the complexity of determining which machine is least loaded. Moreover, it is not useful when compute/communication ratio of the object is small, and does not allow the use of machine-specific names.
- The machine on which the creator of the object is executed. We have chosen this approach in our current implementation because it accommodates programs using machine-specific names. A potential drawback of this approach is that none of the clients communicating with the object may reside on the creator's machine. However, this situation would occur frequently only for those

objects that are equally used by several hosts. We expect the majority of objects, like files, to be used mainly by processes on the creator's machine.

Object Protection

Our initial approach for protecting objects was to let the `execute` right determine if a process can send messages to the protected object. While this approach is simple, it does not let the object protect different kinds of messages independently. Therefore, our current protection scheme is a little more complicated and involves the notion of a *protected port*. A protected port is similar to the regular port supported by the operating system except that it is also associated with an access list. A process may send/receive information from this port only if it has read/write permission to it. Moreover, if a process needs to `connect` to a port created by an object, as in Unix, then it can do so only if it has `execute` access to the object. This two-level protection scheme for ports is inspired by the Unix file protection scheme that requires that a process have `execute` access to a directory before it can `open` a file in it.

Other Operations on Objects

Besides `open`, `open_wait`, `close`, and `object_create`, several other operations may be defined on objects. Objects are extensions of processes, therefore all operations the host system defines on processes such as `kill`, `resume`, and `suspend`, may also be invoked on them. Moreover, since objects are in the file system, some file operations such as `rename`, `create_alias`, and `delete` may also be invoked on them. However, not all file operations supported by the host system are applicable to objects, since some of them such as `read`, `write`, and `seek`, apply only to streams of data (a stream interpretation of objects is discussed in § 5).

4. Implementation

We are currently building a subset of this approach as part of an environment called SUITE (System of Uniform Type-Directed Editors), which is being implemented on a network of hosts running Unix with Sun NFS (Network File system), RPC (Remote Procedure Call), and XDR (eXternal Data Representation). SUITE objects combine not only the program interface of files and processes, but also their user

interfaces. Like files, they can be edited, and like processes, they provide incremental response to user input. More detailed discussion of some of these user interface ideas are available in [6] and [5]

A full implementation of our approach on Unix would have required changes to the kernel to support objects, data files, and protected ports, as special files in the file system. However, we were unwilling to do so in a first-cut experimental implementation of the approach. Therefore we are implementing a subset of the approach that does not require kernel support. We briefly describe below the main features of the implementation. In this discussion we shall differentiate between the the various operations that can be invoked on or by objects. We shall refer to all new operations introduced in § 3 as *object operations*, operations defined on processes by the host system as *process operations*, and operations defined on files as *file operations*. An object may have object, process, or file operations invoked on it.

In SUITE, the process properties of an active object are implemented by an *object process*, which is a normal process that that executes the object's program. A process operation on an active object, such as `kill`, is implemented by invoking the same operation on its object process. Similarly, the file properties of an object are implemented by an *object file*, which is an ordinary file in the system with the same name as the object. A file operation on the object, such as `create_alias`, is implemented by invoking the same operation on its object file.

The object file keeps the active and passive information about the object. The former consists of the identifier of its object process, its reference count, and whether the object has established its active invariant. The latter consists of the object's program name, data file name, status (active or passive), and home (host on which it was created). Data files are ordinary files in the system.

Each host runs an *object manager*, which keeps for every process executing on that machine an *object table* that is indexed by object descriptors and has an entry for each object opened by the process. Each entry contains the name of the object, a file descriptor addressing the object file, and whether the process is waiting for the object to establish its active invariant. A process invokes an operation on an object by calling a library routine that sends an appropriate message (using Sun RPC) to the object manager.

The object manager services a file operation locally by using Sun NFS to access the object file. It services a process operation by forwarding the request to the object manager executing on the object's home, which makes a kernel call to invoke the operation on the object process. The implementations of the object operations simply follow the semantics of these operations described in § 3.

We have implemented initial versions of `object_create`, `open_wait`, `ready`, `close`, `passivate`, and `delete`, with slightly different semantics than the ones described in this paper. We plan to upgrade this version to the planned implementation by June '88.

Our planned implementation, however, does not cover all aspects of our approach. Protected ports are not supported. Moreover, data and object files can be accessed as ordinary files. This implementation can straightforwardly be extended to support the full approach by making the object manager a part of the kernel, and defining object files, data files, and protected ports, as special files in the system.

5. Discussion

In this section, we discuss some distinguishing properties of our approach and consequences of supporting them.

Heterogeneous Objects

Unlike Smalltalk and Eden, our approach allows objects to execute programs in multiple languages. Thus the system does not force the programmer to use any one, possibly unconventional, programming language. On the other hand, single-language systems can offer an integrated programming environment where there is no distinction between system data structures and language data structures. Moreover, they can provide code sharing and classification of objects based on inheritance of classes. It is not clear how such a facility can be provided by a multilanguage operating system.

It is important to note that our approach provides object-based programming only at the operating system level. It does not address programming of the internal data structures of an object, which themselves may be objects if the programming language is object-based. We provide only a way of creating the top-level system data structures as objects. We expect these objects to be used for the same purposes for which files have been traditionally used:

- for keeping data to be shared between processes written in different programming languages and/or executing in different address spaces,
- for keeping data to be protected from access by processes executing on behalf of arbitrary users,
- for keeping persistent data.

It is conceivable that a programmer using a language that does not support objects may also create temporary, private data as top-level objects. For instance, a Pascal programmer may create a `stack` object by defining a server that responds to `push`, and `pop` messages. However, the cost of interprocess communication makes such use impractical.

Moreover, objects can be considered as serious alternatives to files only if the cost of accessing objects can compete with the cost of accessing files. Experiments show that the cost of sending messages to local and remote processes compares favourably with the cost of accessing local and remote files [3, 11]. However, object-access has the added expense of activation and passivation of objects, which involves loading and unloading of the object's program.

We believe the above problem is not so severe since locality of object reference should make activation and passivation of objects infrequent. Moreover, a machine may not actually passivate an object when it calls `passivate`. Instead, it may put it in a cache of object processes that the system looks at before creating a new object process. In case of cache hit, a previously created object process for an object can be reused.

We do not expect all system data structures to be created as objects. For instance, data that has no structure or semantics, such as some mail messages, can be kept in files, and, directory information can be kept in directories. Files, directories, devices, ports, and other resources provided by the operating system can be considered as special objects whose operations are predefined and implemented by the operating system.

Hybrid System

Our approach has adopted the philosophy of extending a conventional systems to support objects. Other systems, such as Hydra, Smalltalk, Eden, and Clouds have taken the opposite approach of creating

new systems based entirely on the concept of objects (that is, all system structures are created as objects). Our approach allows a programmer to experiment with objects without sacrificing a familiar domain. It also creates a system that strictly enhances the existing one. Pure object-based systems, on the other hand, are prototype kernels instead of full systems and do not offer substitutes for the large number of facilities offered by conventional systems.

One disadvantage of our approach is that it is not as uniform as pure object-based systems. Some system data structures are created as objects, while others are created as files, directories, and devices. We have tried to reduce this problem by making objects look like files, thereby providing a common set of file operations such as `rename`, and `delete` for all object and non-object structures accessible via the file system. In this respect, our approach is similar to the one taken by hybrid programming languages, which have tried to make objects look like conventional data structures. However, like these languages, we believe our system is not as uniform as its pure counterparts. For instance, while the `suspend` operation on an appointment server is invoked by making a kernel call, the `addAppt` operation is invoked by sending a message to the server. (This problem is reduced in conventional systems such as Mach, where even kernel calls look like object invocations.)

Objects as Files

Our exercise of integrating objects in a conventional system has lead to us giving objects some unconventional (from the point of view of the object-based systems) properties:

- Objects may need to be opened before access and may be closed after access.
- Objects are addressed by temporary, client-relative descriptors, instead of absolute addresses.
- Objects are protected by access lists instead of capabilities.

We believe these properties are worth exploring even in pure object-based systems. A object-based system may support a protection scheme based on access lists because it is simple and has been successfully used in conventional systems. It may support `open` and `close` for efficiency reasons: The access rights of a client needs to be checked only when it `opens` the object and not every time it accesses it. Invocations of `close` can be used to decide when an object should be deactivated.

On the other hand these properties have some negative consequences. Since descriptors are temporary, object needs to go through the overhead of establishing active and passive invariants, and need to have permanents name in a directory. Since descriptors are object-relative, they cannot be passed in messages to other objects.

These properties are unacceptable in small objects such as integers, since it would be impractical to require, for instance, that each integer have a file name, descriptors be allocated for all integers accessed by an object, and descriptors for integers not be communicated in messages. Therefore integrated programming environments that provide a uniform object model cannot support file properties in objects. However, in systems that do distinguish between small and large objects, we believe it is worth exploring these properties in large objects.

Interestingly, the notion of accessing active entities as files has also been explored in Version 8 Unix [10], which puts processes in the file system, giving them names based on their process identifiers. A debugger can use these names to `open` and `close` processes on its machine and `read` and `write` their code segments. A process in this system is not an object in that it is not persistent or protected from messages sent by arbitrary processes.

Elements of our approach can be combined with Version 8 Unix. The idea of opening and closing remote processes can be introduced in Version 8 to allow distributed debugging of processes. Conversely, the Version 8 `read` and `write` operations can be defined on objects to provide distributed debugging of objects.

6. Conclusions

Keeping system data in files instead of objects leads to at least three problems:

- Files are untyped.
- The syntax and semantics of information stored in files is hard to change.
- The in-core and external copy of data may get inconsistent.

We have presented a simple approach for supporting objects in a conventional system. In comparison to previous approaches, our approach supports heterogeneous objects, provides a hybrid operating system, and gives objects file properties. We are currently implementing an environment called SUITE to explore the usefulness of this approach.

REFERENCES

[1] G.T. Almes, A.P Black, E.D. Lazowska, and J.D. Noe, "The Eden System: A Technical Overview," *IEEE Transactions on Software Engineering SE-11*, January 1985, pp. 43-59.

[2] F. Baskett, J.H. Howard, and J.T. Montague, "Task Communication in DEMOS," *Proceedings of the Sixth Symposium on Operating System Principles*, November 1975, pp. 23-32.

[3] Bharat Bhargava, Tom Mueller, and John Riedl, "Experimental Analysis of Layered Ethernet Software," *Proceedings of the Fall Joint Computer Science Conference*, October 1987, pp. 559-568.

[4] Brad Cox, *Object-Oriented Programming, An Evolutionary Approach*, Addison-Wesley, 1986.

[5] Prasun Dewan, "Automatic Generation of User Interfaces," Ph.D. Thesis and Computer Sciences Technical Report #666, University of Wisconsin-Madison, September 1986.

[6] Prasun Dewan and Marvin Solomon, "Dost: An Environment to Support Automatic Generation of User Interfaces," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices* 22:1 (January 1987), pp. 150-159.

[7] R. A. Finkel, M. L. Scott, W. K. Kalsow, Y. Artsy, H-Y Chang, P. Dewan, A. J. Gordon, B. Rosenburg, M. H. Solomon, and C-Q Yang, "Experience with Charlotte: Simplicity versus function in a distributed operating system," Computer Sciences Technical Report #653, University of Wisconsin–Madison, July 1986.

[8] Adele Goldberg and David Robinson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass., 1983.

[9] Michael B. Jones and Richard F. Rashid, "Mach and MatchMaker: Kernel and Language Support for Object-Oriented Distributed Systems," *OOPSLA '86 Proceedings*, September 1986, pp. 67-77.

[10] T. Killian, "Processes as Files," *Proceedings of the Summer Usenix Conference*, July 1984.

[11] Edward D. Lazowska, John Zahorjan, David Cheriton, and Willy Zwaenepoel, "File Access Performance of Diskless Workstations," *ACM Transactions on Computer Systems* 4:3 (August 1986), pp. 238-268.

[12] Barbara Liskov and Robert Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *Proceedings of the 9th POPL*, 1982, pp. 7-19.

[13] Eugene Spafford, "Architecture and Operation Invocation in the CLOUDS Kernel," Tech. Report No CSD-TR-730, Purdue University, December 1987.

[14] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986.

[15] Peter Wegner, "Dimensions of Object-Based Language Design," *OOPSLA '87 Proceedings*, October 1987, pp. 168-182.

[16] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "Hydra: The Kernel of a Multiprocessor Operating System," *CACM* 17:6 (June 1974).