An Object Model for Conventional Operating Systems

Prasun Dewan Purdue University
Eric Vasilik Sun Microsystems

ABSTRACT: We have developed an object model for conventional (UNIX-like) systems. It can be used for extending such systems with persistent, shared, protected, and distributed objects. It allows objects to coexist with, access, and be accessed by existing components of the operating system, and has been developed by applying much of the work done in naming, organization, access, and protection of conventional resources to support naming, organization, access, and protection of objects.

Objects are created as combinations of conventional processes and files. Like processes, they are active agents capable of executing code on different hosts and communicating with other objects. Like files, they are persistent, have a protected name in a network file system, and are opened and closed for access. The model has been implemented in Suite, which is an extension of UNIX and supports a large part of the functionality provided by unconventional object-based operating systems.

This research was supported in part by the National Science Foundation sponsored Software Engineering Research Center at Purdue.

The paper presents the motivation for the object model, describes its distinguishing features together with the rationale for our decisions, outlines its implementation in Suite, describes the results of our preliminary experience with building and using objects in Suite, and presents conclusions and future directions for research.

1. Introduction

Our interest in objects [Wegner 1987] stems from our work on the Suite user interface software [Dewan 1990a]. Two of the goals of Suite are to support (i) loosely-coupled interactive applications, that is, applications whose interactive and computational components execute in different address spaces residing possibly on different hosts, and (ii) collaborative applications, that is, applications that allow multiple users interacting possibly from different workstations to share results in real-time. Objects facilitate the construction of such applications. Loosely-coupled interactive applications can be constructed by creating their interactive and computational components as separate objects communicating via a high-level remote procedure interface, and collaborative applications can be constructed by creating them as collections of objects interacting with different users and communicating with each other to allow the users to share results in real-time. Figure 1 illustrates how objects may be used to construct a simple loosely-coupled collaborative appointment manager. An appointment object stores the appointments for a particular user and its dialogue manager allows the user to view and modify them. Appointment objects and dialogue managers communicate with each other to allow, for instance, a change made to an appointment by one user to be immediately viewed by other users involved in the meeting.

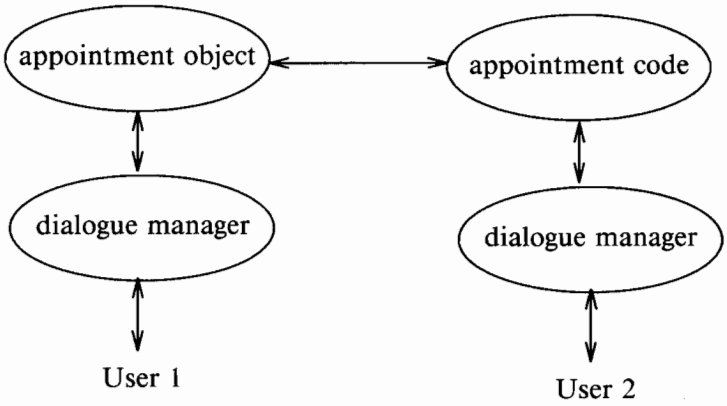

Figure 1: A Loosely-Coupled Collaborative Appointment Manager

None of the existing object models adequately satisfied our needs. Object models supported by programming languages such as Smalltalk-80 [Goldberg & Robson 1983] and C++ [Stroustrup 1986] were unsuitable since they did not support sharing of objects among multiple users. Object models supported by database management systems such as GemStone [Maier 1986] overcome this limitation of language models. However, when we started this project, no distributed object-based DBMS was available. Moreover, object-based DBMS were expected to be too slow for creating high-bandwidth collaborative user interfaces since they translate database references to memory addresses in software.[1] Object models supported by operating systems such as Eden [Almes at al. 1985] and Clouds [Dasgupta et al. 1990] met our requirements best. They support shared, distributed, persistent, and protected objects. Moreover, they map objects to virtual memory, thereby letting the hardware take care of address translation. However, current object-based operating systems have taken the "revolutionary" approach of defining object models that are incompatible with conventional systems. They are new systems built entirely around the concept of objects and offer unconventional methods for supporting operating system

[1]The area of distributed object-based DBMS is still in its infancy and, to the best of our knowledge, there is no experimental data available to confirm our expectation about the performance of such a system. At Purdue, we are currently involved in the engineering of such a system [Dewan et al. 1989] and plan to study its performance for collaborative applications.

components such as devices, 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. Thus, we were unwilling to use any of these systems as a basis for Suite.

Therefore, we developed our own object model which is a variation of the object models supported by Eden and Clouds. Unlike these systems, we have taken the approach of defining a model that integrates objects with conventional systems, in two main ways: First, objects can coexist with, access, and be accessed by existing components of the operating system. In particular, objects can access conventional resources; and conventional processes can invoke methods in objects. As a result, programmers can incrementally explore the use of objects without the fear of sacrificing existing components of conventional systems. Second, we have applied much of the work done in naming, organization, access, and protection of conventional resources to support naming, organization, access, and protection of objects. As a result, few new concepts need to be introduced in the underlying system to support objects and the implementation of existing components in the system can be reused to implement objects.

We have implemented the model in Suite, which is an extension of UNIX supporting TCP/IP and Sun NFS (Network File System) [Sandberg 1986]. In Suite, 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, can be accessed by different users, have a protected name in a network file system, and are opened and closed for access.

While the Suite object layer was designed specifically as an extension of UNIX, it represents a general object model that can be implemented in a conventional operating system to extend the system with distributed, persistent, shared, and protected objects. Indeed, a simpler version of this model was implemented by the first author in Dost [Sweet 1985] on top of the Mesa-based XDE (Xerox Development Environment) [Dewan & Salomon 1987].

The rest of this paper is organized as follows. Section 2 describes the Suite object model, discussing how we handle naming, interobject communication, persistence, activation and

passivation, object placement, and other issues in the design of the model. Section 3 describes its implementation on UNIX. Section 4 outlines our preliminary experience in creating objects in Suite. Finally, Section 5 presents conclusions and directions for future work.

Part of this work was presented previously in Dewan & Vasilik [1989], where we referred to the Suite object layer as DOBS (Distributed Object-Based System).

2. Object model

Before we describe the Suite object model, we need to define a "conventional operating system" and associated terminology. We use this term to refer to an operating system that supports: conventional (Pascal-like) languages; a set of resource-independent file operations (that is, applicable to file and non-file resources) such as `Create`, `Open`, `Close`, `Link`, and `Delete`; a hierarchical multi-user network file system; a fixed number of system-defined access rights that include the `read`, `write`, and `execute` rights; access control based on access lists that keep with each protected resource a list of user groups and their access rights to the resource; UNIX-like user identifiers determining ownership of processes; and `CreateProcess`, `KillProcess`. and other process operations.

2.1 Objects = Files + Processes

In Suite, an object possesses properties of processes and files. Like a process, it executes some program and is associated with an owner and other process properities. The program executed by an object is called its *class*, which is a conventional program without a main procedure, and can be written in one of the languages supported by the system.[2] It contains special comments called *annotations*, which are directives to the *object compiler* used for compiling class declarations. Figure 2 shows a simple C class in Suite.

[2]Currently, our implementation supports only C. However, it has been designed to accommodate other languages, as discussed in § 2.4.

```
/*oc
    Method AddAppointment
    Eternal appt
*/

typedef char *String;
typedef struct { int hour, minute; } Time;
typedef struct { Time start, finish; } Interval;
typedef enum {Yes, No} ApptOK;

typedef struct {
    String with_who;
    Interval when;
    String why;
} Appointment;

typedef struct {
    unsigned num_appointments;
    Appointment *appointments_arr;
} Appointments;

Appointments appts;

ApptOK AddAppointment(appt)
Appointment appt;
{
    ... code for verifying and adding an appointment ...
}
```

Figure 2: An Example C Class

Each instance of this class keeps in the variable `appts` the list of appointments for a user.

Like a file, an object has one or more protected file names, is persistent, and is associated with both an active and a passive state. Its clients can use one of its names in the `OpenObject` call, which returns a temporary *object descriptor* that refers to the object. For instance, a client interested in accessing the object `/usr/joe/appts` may invoke

```
joe_appts = OpenObject("/usr/joe/appts");
```

where *joe_appts* is a variable of the system-defined type OBJECT. It may use this descriptor to send messages to the object (§ 2.2). When it no longer needs to access the object, it can call `CloseObject` to close its connection with the object.

An object may be created by the `CreateObject` call which is a cross between the `CreateProcess` and `Create` calls for creating processes and files respectively. The call 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 permissions for the new object, and returns a descriptor referring to the object. Thus an appointments object may be created by a call of the form

```
joe_appts = CreateObject("/usr/joe/appts", perms,
    "/usr/bin/Appts", host, owner, argc, argv);
```

where `/usr/joe/appts` is the name of the object, `perms` contains the permissions for the object, `/usr/bin/Appts` is the name of the file containing the class of the object, `host` indicates the host on which it is to created, `owner` indicates the owner of the object, and `argc` and `argv` specify arguments for the object. The interpretation of permissions for an object is discussed later.

The `CreateProcess` call provided by the host system can also be used for creating an object. The file name and permissions for the object are specified in special arguments to the program executed by the object. This call allows objects to be created from the host command interpreter, as illustrated below:

```
% /usr/bin/Appts -n /usr/joe/Appts -p 777
```

An object may be deleted from the system by calling `DeleteObject`.

2.2 Communicating with an Object

Objects and processes execute in different address spaces and can exchange information using the IPC channels provided by the host system. In addition, Suite provides a facility for calling high-level *methods* in an object, which is motivated by the research done in remote procedure call [Nelson 1981]. A method is like a procedure except that it can be invoked from remote

address spaces. It is declared using an annotation specifying the procedure that implements it. An invocation of a remote method is like the invocation of a local procedure except that it takes an extra argument specifying the object in which the method is to be invoked. A method is invoked synchronously unless the keyword `Asynchronous` begins the annotation for the method. An asynchronous method must not return a result. Methods can be invoked both by objects and processes.

The class declaration in Figure 2 illustrates how methods are defined in Suite. The class uses the annotation

```
Method AddAppointment
```

to define the synchronous method `AddAppointment`. A client may invoke the method by executing the stub (§ 3)

```
AddAppointment(joe_appts, new_appt);
```

where `joe_appts` is an object descriptor referring to the instance in which the method is to be invoked.

In the above example, the name of the method is identical to the name of the procedure that implements it. While this naming scheme is simple to use, it does not allow an object to invoke a method in another object of the same class, since the local procedure and the (stub for the) remote method have the same names. We resolve this problem by letting the programmer explicitly declare a different name for the method, as illustrated by the following declaration:

```
Method ExternalAddAppointment
          uses AddAppointment
```

It would be useful if remote calls looked the same as local calls since local modules could be be easily replaced with remote objects. Indeed, in the implementation of RPC in Cedar [Birrel & Nelson 1984], local and remote call syntax is identical. As a result, a Cedar procedure call can be bound to a local or remote instance of the module implementing the procedure. However, this approach is not object-based in that it does not allow a procedure call in a client to be directed at multiple user-specified instances of a server module. Since a Suite remote call takes an

extra argument specifying the instance in which the call is to be invoked, it can be used, for example, to inform multiple appointment instances about a new appointment:[3]

```
for (i=0; i < num_instances; i++)
    AddAppointment(appointment_instance[i],
                                    new_appt);
```

2.3 *Persistence*

Suite objects are persistent in that their data structures can be checkpointed on disk and later restored in memory. As a result, changes to their state can survive machine crashes. Moreover, as discussed in § 2.5, they can be passivated in order to release memory resources used by them.

Typically, an object's address space consists of a mixture of persistent and temporary data structures. Therefore, like Argus [Liskov & Schelfer 1982], Suite lets an object specify which of its variables are persistent, as illustrated by the annotation

```
Eternal appts
```

in Figure 2. These data structures are implicitly saved on secondary storage when the object is passivated and restored in its address space when it is activated. Moreover, an object may explicitly checkpoint them at any time by calling `Checkpoint` and later restore them by calling `Restore`.

2.4 *Input/Output*

In Suite, objects may input and output values in three ways: by (i) sending arguments to and receiving results from method invocations, (ii) checkpointing and restoring persistent data structures, and (iii) writing to and reading from terminals, sockets, pipes, and other files. Suite provides a common approach to handle all forms of I/O. Moreover, it supports a "type complete" I/O

[3]Naturally, we could have invented an unconventional Smalltalk-like procedure call mechanism in which even a local procedure call takes an object descriptor such as `self` as an argument. However, that would be inconsistent with our goal of supporting existing conventional programming languages.

system which treats typing and I/O as orthogonal issues and supports I/O of values of arbitrary types including pointers. Thus it generalizes the approach taken by persistent languages [Atkinson & Bruneman 1987] which allow values of arbitrary types to persist. Finally, it supports machine- and language- independent I/O, which is in the spirit of machine- and language- independent interprocess communication supported by Matchmaker [Jones & Rashid 1986] and Sun XDR [Sun 1986].

We define a *generic type-specification language* supporting characters, integers, enumerations, subranges, records, discriminated unions, arrays, sequences (variable length arrays), strings, and pointers. A set of language-specific *transformation routines* convert between the language-specific declarations and their generic counterparts. Moreover, we define a machine-independent *external representation* for the generic data structures. A set of predefined machine-specific *I/O routines* convert between the machine representation of primitive data structures and their external representation. These routines are invoked by I/O routines generated for converting between the internal and external representation of user-defined data structures.

To illustrate, assume that values of type `Appointments` of Figure 2 are to be input/output. Then the C-specific transformation routines are used to generate the following generic type declarations:[4]

```
type
    int = integer range -2147483648
                        .. 2147483647;
    char = character;
    String = string of char;
    Time = record
        hour : int;
        minute : int
    end record;
```

[4]In fact, only an internal symbol table representation of these declarations is generated.

```
Interval =  record
    start : Time;
    finish : Time
end record;
Appointment = record
    with_who : String;
    when : Interval;
    why : String
end record;
Appointments = sequence of Appointment;
```

These declarations together with the machine-specific I/O routines are used to generate I/O routines for converting between the internal representation of values of type `Appointments` and their external representation.

The external representation of a pointer is created by recursively creating the external representation of the data structure to which it refers. If a pointer-connected data structure is output and later input, then the value read is isomorphic to the one that was written.

The external representation of data structures and the associated I/O routines are shared by all three forms of I/O—method invocation, checkpointing/restoring of persistent values, and file I/O. As a result, all three forms benefit from the machine- and language- independence of this representation. For instance, objects executing on different machines can communicate with each other and share files containing structured data. Moreover, it is possible to activate an object on a machine that is different from the one on which it was passivated.[5] The external representation is not suitable for screen input/output since it is not human readable. As described in § 2.9, a generic language-independent dialogue manager is provided for converting between the external and (customizable) visual representations of data structures.

I/O routines are implicitly called during checkpointing/restoring and marshalling/unmarshalling of data. They can also be explicitly invoked to do file I/O. *Complete*

[5]However, for reasons discussed in § 2.8, Suite does not currently support this feature.

annotations are used to specify the types of values that are to be read from and written to files. The object compiler processes them by generating appropriate I/O routines. For instance, the annotation

```
Complete Appointments
```

results in the generation of the I/O routines `ReadAppointments` and `WriteAppointments` which can be explicitly invoked to read and write, respectively, data structures of type `Appointments`.

Suite automatically allocates memory for inputing dynamic data structures such as strings, pointers, and sequences. It also provides type-specific routines for (recursively) freeing these data structures. Moreover, when a method returns, Suite automatically frees memory allocated for storing the arguments unless the method calls `SaveArguments`.

In order to support automatic I/O of a data structure, the system needs to unambiguously determine its type. However, weakly-typed languages such as C do not uniquely type a data structure. For instance, in C, the fields of an undiscriminated C union such as

```
union {
   int f1;
   int *f2}
```

cannot be interpreted uniquely. Similarly, a character pointer such as

```
char *ptr
```

can be used in C as a pointer to a character, a pointer to a null terminated string, or a pointer to an array.

Suite uses several disambiguating rules for weakly-typed data structures in C. It assumes that an enumeration field preceding a union field in a record is the discriminant of the record. Thus it assumes that in the record

```
struct {
   enum {tag1, tag2, tag3} tag;
   union {
      int choice1;
      real choice2;
      int *choice3}}
```

the field *tag* is the discriminant of the union. It also assumes that sequences are simulated by records such as `Appointments` of Figure 2 containing a length field and an array pointer. Moreover, it interprets a character pointer as a pointer to a string. An object can explicitly disambiguate the types of data structures by using special annotations.

An object may manually handle input and output data structures of a particular type by providing a bidirectional *I/O handler* for that type. I/O handlers are in the spirit of Sun XDR routines [Sun 1986] and CLAM bundlers [Cohrs & Miller 1988] for converting between internal and external representations of data. In Suite, they allow objects to input/output values whose types cannot be interpreted uniquely such as undiscriminated unions without preceding discriminant fields.

An alternative approach to handling weakly-typed languages is to require that the types of data structures that are to be input/output be declared directly in the strongly-typed generic language. This approach is taken by Mach [Jones & Rashid 1986] which requires that the types of all parameters/results of remote procedures be declared in the Matchmaker interface specification language. We did not use this approach since it requires that a programmer implementing a class use two different languages.

2.5 Object States and Handlers

Figure 3 shows the various states of an object and handlers called at state transitions. An object may be in the *passive* or *active* state. When an object is passive, a passive representation of it (§ 3) is kept in an *instance file* maintained by the system for the object. Later, when it is activated, this representation is used to create a temporary *instance process* for executing the methods of the object. An active object is associated with a reference count which stores the number of objects that have opened it.

The object creation calls start the object in the active state. The `CreateProcess` call initializes the reference count to 0, while the `CreateObject` call, which also opens the object, initializes the reference count to 1. An `OpenObject` call activates an object if it is passive, and increments the reference count. A `CloseObject` call decrements the reference count of the object. The

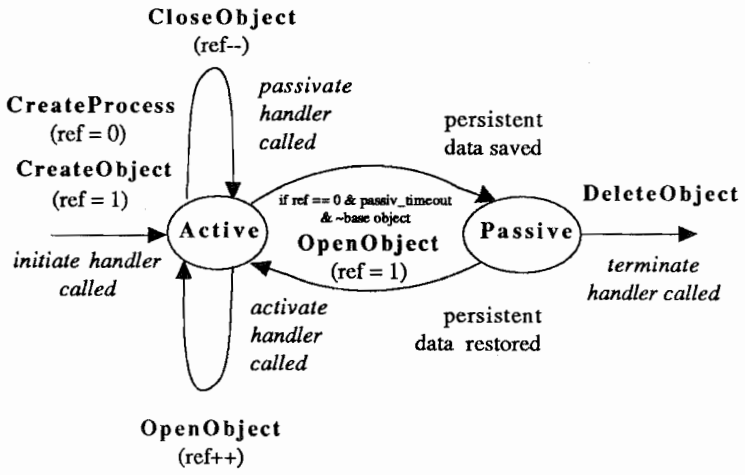

Figure 3: Object State Transition Diagram

`DeleteObject` call removes the object if it is unreferenced, and returns an error message otherwise.

When should an object be passivated? A simple approach (which we implemented initially) is to passivate an object when its reference count goes to zero. Under this approach the cost of opening an unreferenced object or closing the last reference to an object is high since the object has to be activated and passivated, respectively. Our initial measurements indicate that this overhead can be significant. For instance, the combined cost of opening and closing on a Sequent-Symmetry multiprocessor computer a local active object takes 80 milliseconds and a passive object with 40K persistent data takes 2100 milliseconds. Therefore, a better approach is to delay the passivation for a period of time in the hope that the object would be opened soon by some other object, and passivate the object only if it is not accessed during that period. This approach reduces the overhead of opening and closing frequently accessed objects.

However, neither of these approaches supports the notion of *base objects*, that is, objects that need to remain active even when they are not referenced by other objects. These objects, typically, are used to open and manipulate other objects and are not themselves opened by any other object. An example of such an object is a Suite dialogue manager, which allows users to edit other

objects. The dialogue manager is started from the command interpreter and provides a window in which an object can be edited (Figure 5). It supports the "load" "close" and "quit" commands, which are invoked by a user for opening an object for editing, closing the object, and terminating the dialogue manager respectively. A dialogue manager is often not connected to any object and would be deactivated by both approaches described above.

Therefore, we use a modified version of the second approach that provides primitives for supporting base objects. An object can disable passivation by calling `DisablePassivation` and later enable passivation by calling `EnablePassivation`. A base object can disable passivation when it is created or activated and enable passivation when it no longer needs to be active. For instance, a Suite dialogue manager enables passivation when the user executes the "quit" command.

The Suite approach of passivating an object only if its reference count is zero can potentially lead to cycles of "garbage" instance processes doing no useful work. However, in practice, we have found that cycles among instance processes get broken since an initiating event that results in the opening of an object is typically followed by a terminating event that closes the object (§ 4). We have considered, but not implemented, a "mark and sweep" algorithm that only keeps those objects active that are reachable from base objects. Such an algorithm would reduce the likelihood of creating cycles of "garbage" instance processes but not eliminate it since a base object may forget to close an object it refers to.

Before an active object is passivated, it may need to delete windows, close object and file descriptors, and perform other actions necessary for establishing its *passive invariant*. Therefore, Suite lets an object define a *passivate handler* which is invoked before it is passivated.

When a passive object is activated, it may need to create windows, open file and object descriptors, call `DisablePassivation`, and perform other actions necessary to establish its *active invariant*. Therefore, Suite also lets an object define an *activate handler*, which is invoked whenever the object is activated.

A newly created object executes its *initiate handler*, which may be used, for instance, to create an initial set of files, objects, and other resources necessary for the execution of the object. Similarly, before an object is removed, a *terminate handler* is called, which can be used, for instance, to remove the resources being used by the object.

One more handler is defined for supporting periodic background activity in the object. A referenced active object waiting for a request for method invocation periodically executes a *background handler* (Figure 4). The background handler and the time period between invocations of it are specified by the object. This facility can be used by an object to, for instance, periodically refresh its display.

The various state transition handlers and associated parameters are specified in the class of the object by *handler annotations.* For instance, the annotation

```
Background with ApptsBackgroundHandler delay 50
```

in the class of an object specifies the background handler for the object and the time period between invocations of it.

2.6 Class Modules

So far, we have assumed that a class consists of a single module. In general, it consists of several *class modules*, which are like Ada packages. Each class module defines a set of methods and data structures and may be linked with other modules to form a class. Thus a programmer may separately define an `Appts` module and a `Graphics` module and link them together to form the

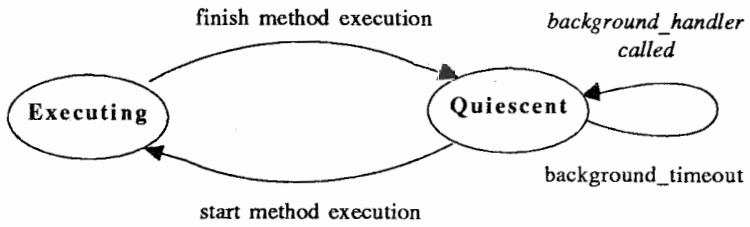

Figure 4: Background Handler

`Graphical_Appts` class. Instances of `Graphical_Appts` would be capable of executing methods defined in both class modules.

Initiate, passivate, and other handlers can be defined in each class module, thereby allowing a class module to define its response to various state transitions independently of other modules. When a state transition occurs in an object, the corresponding handler in each class module defining the object is called.

2.7 Distribution

Objects may reside on different computers connected by a network. For the most part, the design of our object model is independent of whether objects are confined to a single host or distributed. For instance, the syntax of a method invocation is independent of the location of the object in which the method is to be invoked. Two main distribution-related issues in the design of the model are naming and placement of objects.

Our design distinguishes between the naming of objects on same and different LANs. It assumes that each LAN supports a hierarchical network file system allowing a host to name files on other hosts and borrows the naming scheme from the underlying network file system. Thus, Suite inherits the naming scheme from Sun NFS, and lets an object name be put in a directory mounted on a remote system.

This naming scheme has two drawbacks. First, it does not guarantee host-independent object names since the underlying network file system may define local name spaces. In particular, in Suite, it does not allow a dialogue manager on a personal workstation to communicate its name to an object on a shared host, since typically, the file systems of personal workstations are not mounted on shared hosts. This problem would not occur if a network file system such as Andrew [Satyanarayanan 1990] that places names of files in a global space was used. Second, it does not support communication among objects on hosts that are not part of a common network file system. In particular, in Suite, it does not allow us to study the performance of configurations in which dialogue managers and objects execute on hosts connected by a WAN. Therefore, we also support *network-wide* object

names which uniquely identify objects in a network. A network-wide name, like file specifiers supported by the UNIX `rcp` program, specifies the file name of an object relative to a host and is the host name followed by the file name:

```
arthur.cs.purdue.edu: /usr/joe/appts
```

In a distributed persistent object system, the locations of the passive representations and activations of objects need to be determined. In Suite, the passive representation of an object is kept on the host that stores its parent directory, thereby reusing the implementation of name resolution from the underlying file system (§ 3). We initially considered dynamic schemes for object activation that activate an object on different hosts, thereby supporting "object migration". Unlike process migration schemes, these schemes were easy to implement and only required facilities to start instance processes on and read instance files from remote hosts, which existed in our system. Moreover, they were compatible with process migration schemes which could be used to support migration of object activations. Two simple dynamic schemes we considered were:

- Activate an object on the host on which the client that activates it resides. This approach guarantees that at least one client communicating with the object is on the same machine as the object, thereby providing efficient communication between them.[6]
- Activate an object on the host on which the object's instance file resides. This approach supports efficient object activations and passivations since an instance process accesses a local instance file. Moreover, it lets a user migrate objects by renaming them. On the other hand, it is possible for an object to be activated on a machine on which none of its clients reside. This situation would occur frequently when objects are created and opened from diskless workstations.

[6]To illustrate the difference between the costs of invoking local and remote methods, in our implementation, on Sequent-Symmetry multiprocessor computers connected via a 10M bit Ethernet network, the costs of local and remote invocations of synchronous methods taking no arguments and returning no results was 6.7 and 17 milliseconds, respectively.

However, dynamic schemes do not allow an object to use host-specific names, execute on behalf of a fixed user (§ 2.8), or be compiled for a single type of host. Therefore, we currently adopt the simple approach of activating an object on a fixed host in the system specified by the creator of the object, which by default is the host on which the creator resides. It is the creator's responsibility to ensure that the object's class is executable on this machine. A potential problem with this approach is that none of the clients communicating with the object may reside on the machine on which the object executes. This problem is reduced for "private" objects that are accessed mainly by clients executing on the creator's machine.

2.8 Sharing and Protection

Suite objects can access resources belonging to multiple users. Like a conventional process, an object executes on behalf of a specific user called its owner. The owner of an object created locally is the same as the owner of its creator and remotely is the user specified by `CreateObject`.

This approach fixes the owner of the object for its entire lifetime. An alternate approach we considered is to give an object the owner of its activator. Unlike the first approach, the second approach does not require that `OpenObject` communicate with a privileged process to activate objects (§ 3). It was not adopted since it makes the access rights of an object depend on the invocation of `OpenObject` that caused its activation.

Objects are themselves protected resources associated with access lists. The interpretation of these access lists is closely tied to our implementation and is discussed in § 3.

2.9 User Interface

In Suite, we are experimenting mainly with an object-editing user interface, which is a direct extension of text-editing interface provided by conventional systems for manipulating files. It treats all objects as data that can be edited by the user, and checks user changes for syntactic and semantic consistency. Returning to the appointments example, it allows a user to change an appointment

by simply editing a visual representation of the current list of appointments (Figure 5). The user interface of an object is implemented by a generic language-independent dialogue manager, which is itself an object defined by a system-provided class, as discussed later. More details on the user interface can be found in Dewan [1990b].

Other interactive object-based systems such as Eden [Almes et al. 1985] and Hydra [Snodgrass 1983] offer command languages for interactively sending messages to objects. Command-oriented object manipulation complements editor-oriented object manipulation since it allows users to manipulate objects that are not displayed in editor windows. We do not currently provide a special object-manipulation command language since it is possible to write applications programs that are invoked from the command interpreter to send messages to objects. For instance, it is possible to write an application program, `add_appt`, that sends the `AddAppointment` message to an appointment object. This approach increases the overhead of creating a new class of objects since it requires that an application program be written for each method defined by the class. On other hand, it has two important advantages: First, it allows object-manipulation commands to look like existing commands provided by the underlying system, as illustrated below:

```
% add_appt -object ~/appts -user joe \
        -time "12:00" -reason "lunch"
```

Second, it allows aliasing, environment variables, and other primitives provided by the underlying system for easing the invocation of commands to be used for object-manipulation commands. For instance, the `add_appt` command above can make the object parameter optional and use the value of an environment variable as the name of the default appointment object. Nonetheless, it would be useful if a default command language for sending messages to objects was automatically provided by the system.

3. Implementation

When an object is active, it is associated with an *instance process* which executes the object's methods and handlers, checkpoints and restores the persistent data of the object, and keeps its reference count. It is also associated with an *instance file* which keeps the identifier of its instance process, a UNIX port number used for sending messages to it, the name of its class, its home (host on which it was created) and owner, user and group identifiers, and checkpointed data structures.

The operations on objects are implemented as library routines linked to client programs. An instance file is created by the corresponding instance process, read by the `OpenObject` operation, and deleted by the `DeleteObject` operation. It gets the permissions specified for the object in the `CreateObject` and `CreateProcess` calls. As a result, the following protection scheme is defined for objects: An object can create and delete objects in a directory only if it can create and delete files in that directory. It can invoke the `OpenObject` operation on an object only if it has `read` access to the object, since this operation needs to read the instance file.

The name of an instance file is the same as the name of the corresponding object. As a result our implementation uses the underlying implementation of a hierarchical network file system to resolve object names. For instance, the `OpenObject` operation simply opens the corresponding instance file to determine the object to be opened. Moreover, the implementation of file operations such as `Link` and `Unlink` can be directly used for objects. For instance, the operation

```
Link("/usr/joe/appts", "/usr/jack/joe_appts")
```

which creates the new alias `/usr/jack/joe_appts` for the instance file `/usr/joe/appts` also creates the same alias for the corresponding object. Either alias can be used to open the object.

Each host runs an *object manager*, which starts instance processes on that host. A newly created instance process needs to perform several tasks (including creating a port, binding it to a socket, and listening on it) before it can respond to messages from

other objects. Therefore, an object manager blocks the activator of an object until the instance process created for the object indicates that it is ready for receiving messages.

An object and a corresponding instance process can be directly created by the `CreateProcess` call provided by the host system. A subsequent attempt to open the corresponding object may create another instance process if the first process has not finished creating the instance file. Therefore, `CreateProcess` should not be used unless it is certain that the race condition will not occur. This problem does not arise if `CreateObject` is used since creation of instance processes on a host are serialized by the object manager on that host.

An invocation of `OpenObject`, `CloseObject`, or `DeleteObject` that refers to the object by a network-wide name does not directly read the associated instance file. Instead, it communicates with the object manager on the remote host, which accesses the instance file on its behalf. Such an invocation must supply an argument specifying the name of a user on the remote host whose access rights are to be used by the object manager when accessing the instance file. For instance, the invocation

```
OpenObject("arthur.cs.purdue.edu:
                    /usr/joe/appts", "joe")
```

is processed by the object manager on `arthur.cs.purdue.edu` which reads the corresponding instance file with the access rights of user `joe`. An object manager needs to run as a privileged process in order to change its access rights dynamically and set the owner of an instance process.[7]

A class module source is compiled by an *object compiler* which generates *client* and *server* parts for it. The client part contains stubs for invoking methods in the class module while the server part contains the implementation of these methods and support code which includes routines for handling various state transitions. The client part must be linked with any program that invokes methods in the class module and the server part must be

[7]Running it as a privileged process may not be possible on shared hosts. On such hosts, we execute the object manager on behalf of a particular user and make all shared objects world readable and writable.

linked with any class that serves the methods defined in the module. An object compiler is required for each language supported by the system. Our current implementation supports C.

Inter-object communication is built on top of UNIX sockets. An invocation of a method in an object uses the socket layer to send a message to the object. The message contains a module identifier, a method identifier, and the external data representation of the arguments transmitted. The module identifier is used to direct the message towards the *dispatcher* for the module. The dispatcher creates the internal representations of the arguments and invokes the appropriate method in the module. In case of synchronous method invocation it also sends back a return value.

The implementation described above has several limitations:

1. The object operations `OpenObject`, `CloseObject`, `CreateObject`, and `DeleteObject` are implemented as library routines and are thus separate from the corresponding file operations such as `Open`, `Close`, `Create`, and `Delete`.
2. Instance files can be modified in arbitrary ways by processes with appropriate access rights.
3. The protection scheme is tied to the implementation and is unintuitive and coarse grained. For instance, the `read` access right determines if the `OpenObject` operation can be invoked on the object. Similarly, an object cannot separately protect different methods, which is useful for example, to let an appointment object allow certain clients to only read its list of appointments and others to both read and write the appointements.

The first problem can be overcome by making instance files special files in the system and integrating object descriptors with file descriptors and object operations with the corresponding file operations. The kernel can reduce the second problem by ensuring that an instance file is accessed directly only by the corresponding instance process. (This technique ensures that instance files cannot be manipulated by arbitrary processes but does not prevent an object from arbitrarily modifying the corresponding instance file.) The third problem is hard to solve in conventional systems since they support only a fixed number of system-defined access rights. One approach that can be used in

systems such as UNIX that support the `read`, `write`, and `execute` rights is to let an object divide its messages into "read" "write" and "execute" messages protected by the `read`, `write`, and `execute` access rights respectively. Naturally, this scheme is not a good substitute for programmer-defined access rights.

We did not overcome these limitations in our implementation since we were unwilling to modify the kernel.

4. *Preliminary Experience*

We have used the Suite object layer to create new classes of interactive applications not found in traditional systems. We have used objects to support loose coupling between the interactive and computational components of an interactive application. We have developed generic "dialogue managers" which allow users to interact with "editable objects." They display presentations of selected variables of these objects, allow users to edit the presentations in a syntactically and semantically consistent fashion, and communicate with the objects to keep the presentations consistent with the variables they display. Dialogue managers define several remote procedures including `Submit_DM` and `Update_DM` which are invoked by editable objects to display variables and update their displays, respectively.

We have used editable objects and dialogue managers to create several interactive applications. For instance, we have built a "line printer tool" which displays editable listings of line printer queues. A user can open the object using a dialogue manager and edit the display to delete jobs from the system. The line printer tool reads the line printer queues and updates their display periodically as part of its background activity. Similarly, we have built a simple "process tool" which keeps the list of current processes on a particular host. A user can ask a dialogue manager to open a process tool on any system and edit the process list to delete processes from that system. These two tools are representative examples of a whole class of similar tools such as "directory tool" and "current users" tool that allow users to view and modify system data structures.

We have also used objects to built several collaborative applications. We have built a distributed multi-user appointment service consisting of a central "appointment server" and an "appointment filter" for each user of the service. The appointment server maintains the appointments of all users of the service, while an appointment filter displays an aspect of the appointment server consisting of the appointments of a particular user. Each appointment filter interacts with the user via a dialogue manager, which lets the user edit the list of appointments. The appointment server, appointment filters, and dialogue managers communicate with each other to ensure that their copies of the appointment records are kept consistent. Thus, if user A uses a dialogue manager to change his appointment with user B, user B's dialogue manager is informed about the change and updates its display (Figure 5).

The appointment service creates a central database with multiple, distributed views. We have also developed applications that create multiple, distributed databases communicating with each

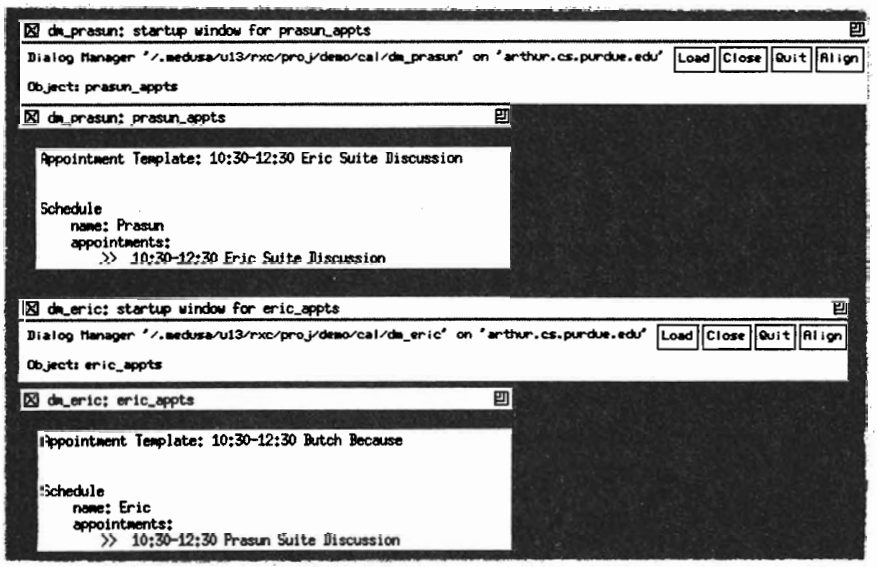

Figure 5: The Appointment Service

other to keep their data consistent. For instance, we have built a multi-user "expense service" which keeps the common expenditures of a group of users. It consists of an "expense" object for each user, which keeps the expenses of all the users and the amount owed to each user. A user can ask a dialogue manager to open his expense object and enter an expense incurred by him. The expense object informs its peers about changes to its user's expenditures, which enter the changes, recalculate the credits, and request their dialogue managers to update the displays.

Similarly, we have built a multi-user integrated project management service for SERC (Software Engineering Center). It consists of the "project" "affiliate" and "budget" objects. The project object lets a user enter the description of a new project, and in response sends a message to the budget object informing it about the new project. Similarly, the affiliate object lets a user enter the name and contribution of a new affiliate, and in response sends a message to the budget object informing it about the new affiliate. The budget object lets a user allocate money to the project, and computes the balance based on the contributions of the affiliates.

Our preliminary experience with the Suite object model has been mostly positive. Persistent objects have proved to be useful, which is illustrated by considering how the applications above would be created in a conventional system. The various data structures would be kept in files, which would be "polled" by conventional processes. To take the project management example, the information about affiliates, projects, and budgets would be kept in files which would be polled by corresponding processes manipulating and displaying this information. The object layer combines the notion of files and processes, thereby supporting active persistent data that can automatically alert other objects interested in changes to it.

The Suite approach to automatic passivation of objects has also proved to be useful. It has been possible to interact with several applications successively without the fear of either (1) creating a large number of instance processes, some of which, such as the line printer and process tools, can put a heavy load on the system, or (2) frequent passivations/activations since referenced objects and unreferenced objects that are frequently accessed are

kept active. Moreover, in our experience, a sequence of initiating events that causes cyclic references among instance processes is followed by a sequence of terminating events that deletes these references. To illustrate how cyclic references are created and broken in Suite, consider what happens when the user executes the "load" command in a dialogue manager to connect to an editable object: (1) The dialogue manager opens the object and executes the method `LoadDialog` in it. (2) `LoadDialog` opens the dialogue manager, thereby causing a cycle. Conversely, when the user executes the "close" command in the dialogue manager, the following events takes place: (1) The dialogue manager closes the object and executes `UnLoadDialog` in it. (2) `UnLoadDialog` closes the dialogue manager, thereby breaking the cycle. Thus with careful programming it has been possible to ensure that cycles of "garbage" instance processes are not created.

Type-completeness of the I/O system has also proved to be useful, since it has relieved programmers from the tedious work of converting between internal and external representations of data structures. Moreover, a uniform I/O system integrating all forms of I/O has been simple to understand and use since it provides, for instance, one set of rules to disambiguate weakly-typed data.

Programmers have benefited in many ways from integration of the Suite object layer with the underlying UNIX layer. They have been able to use a familiar environment consisting of the UNIX system and the C programming language. Moreover, they have perceived the object layer as a natural extension of the existing system. As a result, they have exhibited little resistance to exploring a new paradigm and have been able to learn and use the system in a matter of days. Furthermore, they have been able to reuse services of existing tools. For instance, the process tool executes the UNIX "ps" command to get the current status of the printer queue.[8]

The simple protection scheme implemented in Suite has proved to be sufficient for some of the applications built by us. For instance, a user's line printer tool is made `self` readable, thereby allowing dialogue managers started by the user to invoke

[8]Unfortunately, the tool "polls" the status of the process queue in order to display current information. Changes in the kernel would be required to asynchronously inform the tool whenever process information changed.

methods such as `DeleteJob` in the object. Similarly, an expense object is made `group` readable, where the group consists of the group of users sharing the common expenses. On the other hand, it has proved to be restrictive for other applications. For instance, a budget object cannot allow certain users to read its contents while allowing other users to also modify them. Similarly, an appointment server cannot allow an appointment filter for a user to change only the appointments for that user. The first problem would not occur if we had been willing to change the kernel to separate read and write methods. The second problem would not occur if the underlying system allowed the receiver of a message to authenticate the sender.[9]

An important factor in the usability of an object model is the performance of its implementation. In our prototype implementation, we made no special effort to get good performance. Nonetheless, the response times of most of the applications we built were acceptable, that is, there were no perceptible delays in interacting with them. The exceptions were the process and line printer tools, which drove up the response time of the system by polling the system for the current status of the process list and line printer queue, respectively. These problems have less to do with our approach and more to do with our unwillingness to change the kernel and other tools (such as the line printer daemon on UNIX) to inform interested objects about changes to data structures maintained by them. Indeed, as mentioned above, one of the advantages of supporting objects is that polling of data encapsulated by them is unnecessary.

More details on our experience with the Suite object layer can be found in Dewan & Goudhary [1989].

5. *Conclusions and future work*

The Suite object model was developed to support loose coupling between the interactive and computational components of interactive applications, and collaborative applications. It provides persistent distributed, shared, and protected objects, and has been

[9]or if we had been willing to communicate encrypted messages.

implemented as on top of UNIX, TCP/IP, and Sun NFS (Network File System) [Sun 1986]. It has been used to build several prototype applications that would have been difficult to build in conventional systems. Overall, our preliminary experience with it has been positive.

Several properties of Suite are present in other object-based systems. For instance, persistent and multi-user objects are also supported by several other systems including Hydra [Wulf et al. 1974], Argus [Liskov & Scheifler 1982], Eden [Almes et al. 1985], Clouds [Dasgupta et al. 1990], and GemStone [Maier et al. 1986]; distributed objects communicating via remote procedure calls are supported by most modern systems including Mach [Jones & Rashid 1986], Andrew [Satyanarayanan et al. 1990], Hermes [Black & Artsy 1990], Argus, Clouds, and Eden; protected objects are supported by capability-based systems such as Hydra, Eden, and Clouds; and activation and passivation of objects is supported by Eden and Clouds. The novel aspects of the Suite object layer include: (i) treating objects as combinations of conventional processes and files; (ii) activation, passivation, and other state-transition handlers; (iii) an object passivation scheme based on the number of active references to objects; (iv) support for existing conventional programming languages; and (v) a type-complete I/O system providing a common approach for marshalling/unmarshalling arguments/results of method invocations, checkpointing/restoring persistent values, and writing to/reading from sockets, pipes, and other files.

Our approach has at least three drawbacks: First, since objects, like processes, access resources through temporary descriptors, they need to establish active and passive invariants. This overhead is not necessary in capability-based systems such as Hydra, Eden, and Clouds, in which objects access all resources through persistent capabilities that are valid from one activation to another. Second, objects and other resources are protected by conventional access lists instead of capabilities. Capability-based systems offer a more sophisticated protection mechanism which can ensure, for instance, that different objects created by the same user have access to different sets of resources. Third, objects cannot be used to encapsulate small data structures such as integers and stacks, since it is impractical, for instance, to associate an

integer with its own address space and file name. Objects models supported by most object-oriented languages support small, private, temporary objects and database management systems support both small, private, temporary and large, shared, persistent objects.

The main benefit of our approach is that it supports integration of objects with conventional systems, in two main ways. First, objects can coexist with, access, and be accessed by existing components of the operating system. In particular, objects can access conventional resources such as pipes, sockets, and files; and conventional processes can invoke methods in objects. As a result, programmers can incrementally explore the use of objects without the fear of sacrificing existing components of conventional systems. Second, we have applied much of the work done in naming, organization, access, and protection of conventional resources to support naming, organization, access, and protection of objects. As a result, few new concepts need to be introduced in the base system to support objects and the implementation of existing components in the system can be reused to implement objects.

While the Suite object layer was developed to meet two specific user interface requirements, it has broad applications. In general, it can be used to replace passive files and directories with distributed "active databases" [Morgenstern 1983]. We plan to use it to explore replacement of existing files and directories in our system with corresponding objects. We also plan to expand our set of collaborative applications to include, for instance, a collaborative multi-module program editor.

We have so far not addressed two important issues in the design of objects: transactions and class evolution. These have been addressed, for instance, by Argus and ORION [Banerjee et al. 1983], respectively, and we plan to explore how current approaches to addressing them can be integrated with the Suite object model.

Acknowledgments

Paul Thomas built an early version of the object manager. Rajiv Choudhary, Paul Buis, Harry Duin, Joe Heim, and Harlene Sepulveda helped us test the current implementation of Suite. Harlene Sepulveda and Harry Duin built the line printer tool. Rajiv Choudhary implemented network-wide naming of objects and made performance measurements. Harry Duin, Ronnie Miller, John Riedl, Harlene Sepulveda, and the referees gave useful suggestions for improving the presentation of the paper.

References

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

Malcolm P. Atkinson and O. Peter Buneman, Types and Persistence in Database Programming Languages, *ACM Computer Surveys* **19**(2) (June 1987).

Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth, Semantics and Implementation of Schema Evolution in Object-Oriented Databases, *Proceedings of ACM SIGMOD Conference on Management of Data* (March 1987).

Andrew D. Birrel and Bruce Jay Nelson, Implementing Remote Procedure Calls, *ACM TOCS* **2**(1) (February 1984).

Andrew P. Black and Yeshayahu Artsy, Implementing Location Independent Invocation, *IEEE Transactions on Parallel and Distributed Systems* **1**(1) (January 1990).

David L. Cohrs and Barton P. Miller, Distributed Upcalls: A Mechanism for Layering Asynchronous Abstractions, *IEEE International Conference on Distributed Computing Systems*, pages 55-62 (June 1988).

P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. F. Appelbe, J. M. Bernabeu-Auban, P. W. Hutto, M. Y. A. Khalidi, and C. J. Wilkenloh, The Design and Implementation of the Clouds Distributed Operating System, *Computing Systems* **3**(1), pages 11-46 (Winter 1990).

Prasun Dewan and Eric Vasilik, Supporting Objects in a Conventional Operating System, *Proceedings of the San Diego Winter '89 Usenix Conference*, pages 273-286 (February 1989).

Prasun Dewan, Ashish Vikram, and Bharat Bhargava, Engineering the Object-Relation Database Model in O-Raid, *Proceedings of the Third International Conference on Foundations of Data Organization and Algorithms*, pages 389-403, Springer Verlag (June 1989).

Prasun Dewan and Marvin Solomon, An Approach to Support Automatic Generation of User Interfaces, *ACM Transactions on Programming Languages and Systems* **12**(4), page 566-609 (1990). Preliminary version presented at the *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices* 22:1 pp. 150-159 (January 1987)

Prasun Dewan, A Tour of the Suite User Interface Software, *Proceedings of the 3rd ACM SIGGRAPH Symposium on User Interface Software and Technology*, page 57-65 (1990).

Prasun Dewan and Rajiv Choudhary, Experience with the Suite Distributed Object Model, *Proceedings of IEEE Workshop on Experimental Distributed Systems*, page 57-63 (1990).

Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation,* Addison-Wesley, Reading, Mass. (1983).

Michael B. Jones and Richard F. Rashid, Mach and MatchMaker: Kernel and Language Support for Object-Oriented Distributed Systems, *OOPSLA '86 Proceedings*, pages 67-77 (September 1986).

Barbara Liskov and Robert Scheifler, Guardians and Actions: Linguistic Support for Robust, Distributed Programs, *Proceedings of the 9th POPL*, pages 7-19 (1982).

David Maier, Jacob Stein, Allen Otis, and Alan Purdy, Development of an Object-Oriented DBMS, *OOPSLA '86 Proceedings*, pages 472-483 (September 1986).

Sun Microsystems, Inc., External Data Representation Protocol Specification, *Networking on the Sun Workstations* (1986).

M. Morgenstern, Active Databases as a Paradigm for Enhanced Computing Environments, *Proceedings of the 9th International Conference of Very Large Data Bases*, pages 34-42 (1983).

Bruce Jay Nelson, Remote Procedure Call, Ph.D. Thesis and Tech Report CMU-CS-81-119, Department of Computer Science, Carnegie-Mellon University (May 1981).

R. Sandberg, The Sun Network File System: Implementation and Experience, *Proceedings of the Florence Spring '86 EUUG Conference* (1986).

Mahadev Satyanarayanan, Scalable, Secure, and Highly Available Distributed File Access, *IEEE Computer* **23**(5), pages 9-22 (May 1990).

R.T Snodgrass, An Object-Oriented Command Language, *IEEE Transactions on Software Engineering* **SE-9**(1), pages 1-7 (January 1983).

Bjarne Stroustrup, *The C++ Programming Language,* Addison-Wesley, Reading, Mass. (1986).

Richard E. Sweet, The Mesa Programming Environment, *Proceedings of the ACM SIGPLAN Symposium on Language Issues in Programming Environments*, pages 216-229 (June 1985).

Peter Wegner, Dimensions of Object-Based Language Design, *OOPSLA '87 Proceedings*, pages 168-182 (October 1987).

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).

[*submitted Dec. 27, 1989; revised Sept. 20, 1990; accepted Oct. 3, 1990*]