Distributed Object-Based Programming Systems

ROGER S. CHIN AND SAMUEL T. CHANSON

Department of Computer Science, University of British Columbia, Vancouver, B.C., Canada V6T 1W5

The development of distributed operating systems and object-based programming languages makes possible an environment in which programs consisting of a set of interacting modules, or objects, may execute concurrently on a collection of loosely coupled processors. An object-based programming language encourages a methodology for designing and creating a program as a set of autonomous components, whereas a distributed operating system permits a collection of workstations or personal computers to be treated as a single entity. The amalgamation of these two concepts has resulted in systems that shall be referred to as *distributed, object-based programming systems.*

This paper discusses issues in the design and implementation of such systems. Following the presentation of fundamental concepts and various object models, issues in object management, object interaction management, and physical resource management are discussed. Extensive examples are drawn from existing systems.

Categories and Subject Descriptors: C.2.4 [**Computer-Communications Network**]: Distributed Systems; D.4 [**Software**]: Operating Systems; D.4.1 [**Operating Systems**]: Process Management; D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.6 [**Operating Systems**]: Security and Protection.

General Terms: Design, Reliability, Security

Additional Key Words and Phrases: capability scheme, distributed operating systems, error recovery, method invocation, nested transaction, object-based programming languages, object model, object reliability, processor allocation, resource management, synchronization, transaction

1. FUNDAMENTAL CONCEPTS

1.1 Objects

An *object* is an entity that encapsulates some private *state information* or data, a set of associated *operations* or procedures that manipulate the data, and possibly a thread of control so that collectively they can be treated as a single unit (Figure 1). In general, an object's state is completely protected and hidden from all other objects. The only way it can be examined or modified is by making a request or an *operation invocation* on one of the object's publicly accessible operations. This creates a well-defined interface for each object, enabling the specification of an object's operations to be made public while at the same time keeping the implementation of the operations and the

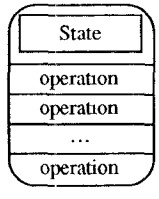


Figure 1. Object.

CONTENTS

representation of its state information private.

An operation may invoke other operations, possibly on other objects. These operations may in turn make invocations on others, and so on. A chain of related invocations is referred to as an *action* (Figure 2).2

1.2 Object-Based Programming Languages

A *class* is the direct extension of the notion of an abstract data type; it is a template from which objects may be created. Every object is an *instance* of some class.[1] A group of objects that have the same set of operations and the same state representations are considered to be of the same class. It should be emphasized that a class maintains no state and performs no operations. Classes typically exist at compile time; objects exist at execution time.

A programming language is defined as being *object based* if it supports objects as a language feature and *object oriented* if it also supports the concept of *inheritance*.[2] Inheritance is a mechanism that permits new classes to be developed from existing classes simply by specifying how the new classes differ from the originals. A class may inherit the operations and behavior of a *base class* or *superclass*, and it may have its operations and behavior inherited by a *derived class* or *subclass*. More than one class may be derived from a single base class. A superclass provides functionality common to all of its subclasses, whereas a subclass provides additional functionality to specialize its behavior. The association between these classes is sometimes referred to as an IS-A or an IS-A-KIND-OF relationship. For example, the superclass dog may have the two subclasses Saint Bernard and dachshund. The inheritance scheme is advantageous in two respects. First, by altering an attribute of a base class, that attribute is similarly modified in all of its derived classes and in all of their subclasses. Second, it encourages the reuse of existing code.

Inheritance from a single base class is referred to as *single inheritance*, whereas inheritance from multiple base classes is referred to as *multiple inheritance*. Another variation of the inheritance scheme is *delegation*. Delegation is a mechanism that permits an instance object to delegate responsibility for servicing an invocation request to another object. This differs from the inheritance scheme in that it is class independent. Individual instance objects of the same class may

[1] A class is sometimes inappropriately referred to as a class object, whereas its instances are referred to as instance objects. In this paper, the term *object* will be used in the context of instance objects.

[2] A detailed discussion of terms that can be used to describe object-oriented languages can be found in Wegner [1987].

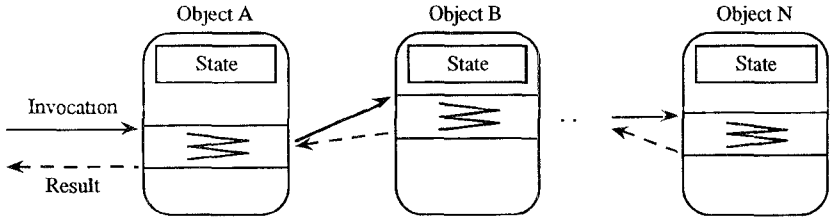


Figure 2. Action.

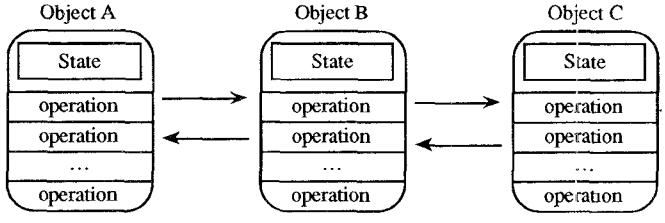


Figure 3. Object-based program.

have different objects servicing requests they are unable to service.

According to these definitions, the languages C + + [Stroustrup 1986], SR [Andrews 1988], and Smalltalk [Goldberg 1983] are object-oriented languages, whereas the languages Ada [DOD 1980] and Modula-2 [Wirth 1985] are object-based languages.

Object-based programming languages encourage their users to design and develop a program consisting of multiple, interacting, autonomous objects (Figure 3). This philosophy of dividing a program into multiple subcomponents is not new. In fact, many of these ideas come from traditional software engineering principles that stress such a design methodology. According to these principles, a program should have the following five characteristics:

- *Abstract.* The design concepts should be separated from the implementation details. A program should hide the design decisions and the data structures used.
- *Structured.* A large program should be decomposed into components of a manageable size, with well-defined relationships established between the components.
- *Modular.* The internal design of each component should be localized so that it does not depend on the internal design of any other component.
- *Concise.* The code should be clear and understandable.
- *Verifiable.* The program should be easy to test and debug.

An object-based programming language emphasizes such a style of programming.

Object-based programming languages can simplify the task of programming because they allow the abstractions used by a programmer to be more easily translated into the abstractions provided by the language. These languages stress a development methodology that emphasizes modular structure, data abstraction, and code reuse. These characteristics loosely correspond to the human approach of dividing a complex problem into less complex subproblems, of focusing on the higher level issues and ignoring those details that are initially of little or no concern, and of drawing upon previously obtained knowledge, respectively. Object-based programming

languages help simplify the task of translating a problem into a program by creating a closer coupling between the programming language and the programmer.

1.3 Object-Based Programming Systems

An *object-based programming system* (*OBPS*) can best be defined as a computing environment that supplies both an object-based programming language and a system that supports object abstraction at the operating system level.[3] This enables objects to be maintained, managed, and used efficiently. It also permits objects to be shared by multiple users; in contrast, object-base programming languages do not allow objects to be shared. To provide the later functionality, the system typically assigns a unique identifier to each object so each can be uniquely specified.

The distinction between an operating system and the programming language it supports is not as great as it once was. In 1976, Jones [1976] observed that operating systems and programming languages were once designed and developed independently, but a close coupling was beginning to form between them. More recently, languages are providing features that were previously provided only by the operating system and vice versa. One reason for this tight coupling is so that efficient low-level support can be provided for the higher level abstractions. This is part of the process that Nicol et al. [1987] call *total system design*. It is their opinion that when a new computing environment is being developed, the language, the operating system, and possibly the hardware of the system should all be designed simultaneously. Each component can then be built to support a particular environment so a more uniform and efficient system is created. One disadvantage to this scheme is that it may sacrifice future flexibility by tying a system to particular abstractions.

The operating system of an OBPS supplies a global, machine-wide object space by providing features for the following:

- Object management
- Object interaction management
- Resource management

These topics are discussed in detail in subsequent sections of this paper.

1.4 Distributed, Object-Based Programming Systems

A *distributed, object-based programming systems* (*DOBPS*) provides the features of an object-based programming system as well as a decentralized or distributed computing environment (see Figure A1). DOBPSs typically have the following characteristics:

- *Distribution.* A DOBPS combines a network of independent, possibly heterogeneous, workstations or personal computers (hereafter referred to as workstations) so that they provide a decentralized computing environment.
- *Transparency.* The system may hide the distributed environment or other underlying details from the users. For example, a DOBPS can provide the feature of *location transparency* so a user does not have to be aware of the machine boundaries and the physical locations of an object in order to make an invocation on it. It should also provide uniform access to all of the objects of the system, whether they are active in memory or inactive in secondary storage.
- *Data Integrity.* A DOBPS ensures that a persistent object is always in a valid state before it performs an invocation. That is, an object is always in a state that is the result of the successful termination of an operation. If an operation does not successfully complete, the

[3]Implementing inheritance at the operating system level is a task that is in need of much more research. Thus, for generality, the scope of this paper will be limited to describing object-based programming systems rather than object-oriented programming systems.

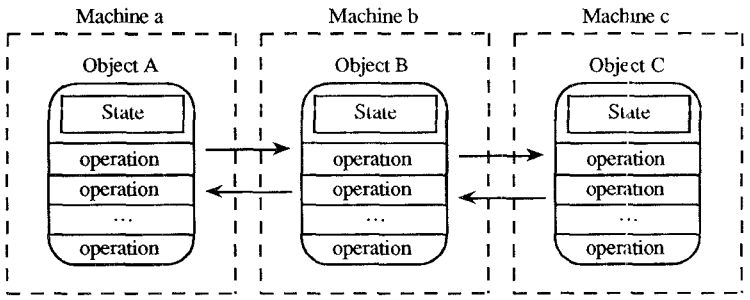


Figure 4. Distributed object-based program.

system ensures that all changes made to the object's state are undone.

- *Fault Tolerance.* The failure of a workstation or an object represents only a partial failure to a DOBPS; the loss is restricted to that workstation or object. The remainder of the system should be able to continue processing with perhaps the inconvenience of a less than normal service.
- *Availability.* A DOBPS may take steps to ensure that all objects remain available to a high probability, despite workstation failures. If any service becomes inaccessible, the entire system may be shut down and restarted in order to restore full service.
- *Recoverability.* If a workstation fails, a DOBPS automatically recovers the persistent objects that resided on it.
- *Object Autonomy.* A DOBPS may permit the owner of an object to specify the clients that have the authority to make invocations on the object.
- *Program Concurrency.* A DOBPS should be able to assign the objects of a program to multiple processors so they may execute concurrently (Figure 4).
- *Object Concurrency.* An object should be able to serve multiple, nonmodifying invocation requests concurrently. Note that this is not true concurrency unless an object resides in a multiprocessor, since only one request can be processed at any one time.
- *Improved Performance.* A well-designed program can typically execute more quickly in a DOBPS than in a conventional system.[4]

Some of these topics will be discussed in greater detail in subsequent sections of this paper.

2. OBJECT STRUCTURE

The structure of the objects supported by a DOBPS influences its overall design. This section defines three types of objects that can be supported by DOBPSs and two ways they can be composed. It then provides a brief overview of how objects are structured in a number of existing systems.

2.1 Granularity

The relative size, overhead, and amount of processing performed by an object characterizes its *granularity*. In the simplest case, a DOBPS supports only *large-grain objects*. These objects are characterized by their large size, relatively large number of instructions they execute to perform an invocation, and relatively few interactions they have with other objects. Some examples of a large-grain object are a major component of a program, a file, and a single-user database. Large-grain objects typically reside in their own address spaces. This enables a system to provide hardware

[4]Guidelines for writing well-designed object-oriented programs are described in Lieberherr and Holland [1989].

protection between objects and ensures that a software fault is contained within the responsible object, unless it is a catastrophic failure. The pure large-grain object scheme offers the advantage of simplicity, but there are a number of drawbacks associated with a DOBPS that supports this scheme. First, large-grain objects are very heavy-weight entities since a separate address space is provided for each object. Second, the system's control and protection over data are at the level of the large-grain objects. This restricts the flexibility of the system and the amount of object concurrency that can be provided. Finally, the system does not provide a consistent data model. Larger data entities are represented as objects, whereas smaller data entities such as linked lists or integers are represented as conventional programming language data abstractions.

To provide a finer level of control over data, a DOBPS may support both large-grain and *medium-grain objects.* Medium-grain objects can be created and maintained relatively inexpensively because they are smaller in size and in scope than large-grain objects. Typical examples of medium-grain objects are data structures such as a linked list, a queue, and the components of a multiuser database. A number of medium-grain objects may reside in the address space of a single large-grain object. This permits a large-grain object to have a greater amount of concurrency, since synchronizing access to the data (see Section 3.2) may be done at the level of the medium-grain objects. Similarly, the amount of data copied to secondary storage when an action commits (see Section 3.1) can be reduced. A drawback of using medium-grain objects is the additional overhead due to the greater number of objects that have to be managed by the system. In addition, a consistent data model is still not provided.

To provide an even finer level of control over data, a DOBPS may support large-grain, medium-grain, and *fine-grain objects.* Fine-grain objects are characterized by their small size, small number of instructions they execute, and relatively large number of interactions they have with other objects. Some examples of fine-grain objects are data types that are provided by conventional programming languages such as Booleans, integers, and complex numbers. Each fine-grain object is encapsulated by and resides within the address space of a single medium-grain or large-grain object. Poor performance due to the overhead of managing a huge number of objects and for making an object invocation for almost every operation is the major problem of systems using fine-grain objects. This approach does, however, provide a completely uniform environment and a consistent data model such that every data entity, no matter how large or small, is an object.

2.2 Composition

The relationship between the processes and the objects of a DOBPS characterizes the composition of the objects. The processes may either be separate and temporarily bound to the objects they invoke, or they may be coupled and permanently bound to the objects in which they execute. These two approaches correspond to the passive object model and the active object model, respectively.

Allowing multiple server processes to execute concurrently within an object permits it to service multiple invocation requests concurrently. The amount of object concurrency that actually takes place, however, will depend on the type of synchronization scheme supported by the system (see Section 3.2) and the particular invocation requests received.

2.2.1 Passive Object Model

In the *passive object model,* the processes and objects of a DOBPS are completely separate entities. A process is not bound nor is it restricted to a single object. Instead, a single process is used to perform all the operations required to satisfy an action. Consequently, a process may execute within several objects

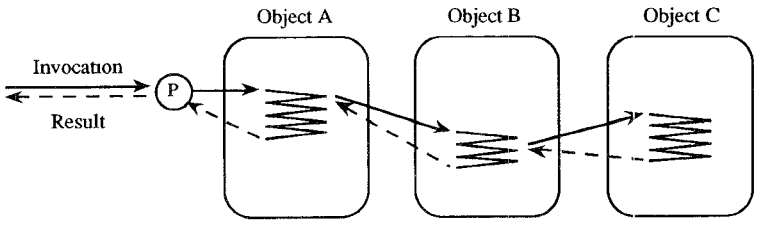


Figure 5. Performing an action in the passive object model.

during its lifetime. When a process makes an invocation on another object, its execution in the object in which it currently resides is temporarily suspended. Conceptually, the process is then mapped into the address space of the second object, where it executes the appropriate operation. When the process completes this subsequent operation, it is returned to the first object, where it resumes the execution of the original operation (Figure 5). A detailed description of this interaction is given in Section 4.2.

One advantage to using the passive object model is that there is virtually no restriction on the number of processes that can be bound to an object. Drawbacks of the passive object model are that the cost of mapping a process into and out of the address space of multiple objects can be difficult and expensive.

2.2.2 Active Object Model

In the *active object model*, several server or worker processes are created for and assigned to each object to handle its invocation requests. Each process is bound and restricted to the particular object for which it is created. When an object is destroyed, so are its processes. In the active object model, an operation is not typically accessed directly by the calling process, as in the case of a traditional procedure call. Instead, when a *client* makes an operation invocation, a process in the corresponding *server* object accepts the request and performs the operation on the client's behalf. The interactions between a client and a server are as follows:

(1) The client presents an invocation request, together with a list of arguments, to the appropriate object specifying the operation to be invoked.
(2) The server object accepts the request, then locates and performs the specified operation.
(3) If in the course of executing an operation an invocation on another object is made, the process issues the invocation request and waits for a result. A server process in the second object is then called upon to execute the new operation, and so on.
(4) When the operation completes, the server returns a result to the client.

In this approach, multiple processes may be involved in performing a single action (Figure 6).

In the *static* variation of the active object model, a fixed number of server processes are created for each object that is created or activated (see Section 5.1). When a request is delivered to an object, it is randomly assigned to an idle server process, which performs the specified operation. Requests delivered to an object whose processes are all busy executing other tasks are placed in a queue of pending requests and serviced at a later time. In this scheme, the maximum degree of object concurrency is further limited by the number of server processes created for each object.

In the *dynamic* variation of the active object model, server processes are

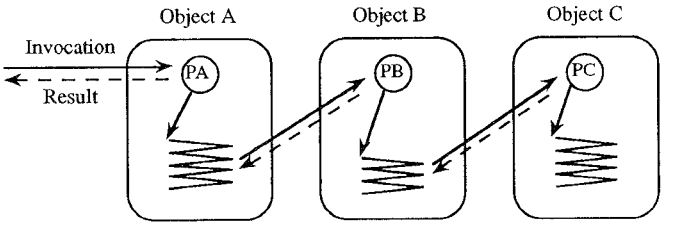


Figure 6. Performing an action in the active object model.

dynamically created for an object as required. Requests are never queued. When a request is delivered to an object, a new process is created for the object to service the request. When the execution of an operation completes, the process that was performing the task is destroyed. The dynamic variation of the active object model has the additional expense of dynamic process creation and destruction. This may be minimized by, for example, maintaining a pool of idle processes.

One problem with the active object model is deadlock. Deadlock can occur if an object does not have enough server processes to handle the requests delivered to it. For example, a single action may invoke the same object more times than there are server processes (e.g., within a recursive invocation, which may span multiple objects). A server process will be assigned to handle each of the invocation requests until no more server processes are available, at which point both the object and the action are blocked. Deadlock is less of a problem in the dynamic variation of the active object model; however, it may still occur if an action is permitted to make an arbitrary number of invocations.

2.3 Overview of Object Structures in Existing Systems

2.3.1 Amoeba

Amoeba [Mullender and Tanenbaum, 1985, 1986; and van Renesse 1987; Tanenbaum et al. 1986] supports large-grain objects and the static variation of the active object model. Large-grain objects are composed of a number of code and data segments that may be shared by multiple objects.

2.3.2 Argus

Argus [Liskov 1988; Liskov et al. 1988; Oki et al. 1985; Walker 1984] supports both large-grain and medium-grain objects, and the dynamic variation of the active object model. The expense of dynamic process creation and destruction is reduced by maintaining a pool of unused processes. When a new process is required, it is removed from the pool. When a process is no longer required, it is returned to the pool. A new group of processes is created only when the pool is emptied.

2.3.3 CHORUS

CHORUS [Banino et al. 1985; Guillemont and Martins 1987; Rozier and Martins 1987] supports large-grain objects and the static variation of the active object model. It differs from most DOBPSs in that only a single server process is created for each object. Furthermore, a server process cannot be interrupted nor can it be blocked while it is executing an operation. All invocations that are made by a process while it is executing an operation are recorded in a transmit queue. These requests are issued only when the operation completes.

2.3.4 Clouds

Clouds [Ahamad and Dasgupta 1987; Ahamad et al. 1987; Dasgupta 1986; Dasgupta et al. 1988; Pitts and Dasgupta,

1988; Spafford 1987] supports large-grain objects and the passive object model. An object consists of a number of segments, including a code segment, one or more data segments for persistent data, and a number of heap segments for volatile data. Clouds permits code segments to be shared by multiple objects. When an invocation is made on an object, the associated process enters the object through one of the entry points in the code segment, executes the corresponding operation, then leaves the object. Special paging hardware is used to map objects into the address spaces of processes efficiently (see Section 5.3).

2.3.5 Eden

Eden [Almes et al. 1985; Pu et al. 1986] supports large-grain objects and the static variation of the active object model. Each object has three components: a long-term state that contains the persistent information, a short-term state that contains the volatile information, and a code segment that contains the operations. An object also contains a number of processes, including a Dispatcher process, one or more server processes, and possibly some maintenance processes. Each Dispatcher process is responsible for accepting invocation requests delivered to its object and assigning them to idle server processes.

2.3.6 Emerald

Emerald [Black et al. 1986a, 1986b] is both a system and a language that supports the passive object model. The Emerald language supports large-grain, medium-grain, and fine-grain objects. The kernel, however, only supports large-grain and medium-grain objects. Fine-grain objects are invisible to the kernel and are incorporated directly into the code of the other objects by the compiler. Large-grain objects may be invoked by any object in the system. They are also the smallest entities that can be moved independently about the system (see Section 5.2). Medium-grain and fine-grain objects are not visible outside of the object in which they are contained; they may be invoked only by objects that reside in the same containing object.

A process consists of a stack composed of a collection of *activation records*. A new activation record is created and added to a process' stack whenever the process makes an invocation on a new object. Each record stores information regarding the state of the process within the object to which it corresponds.

Emerald differs from most DOBPSs in that a single, common address space is shared by all objects and processes that reside in the same workstation. This lowers the cost of creating and maintaining objects and enables objects in the same workstation to examine and modify each other directly (without context switches). A drawback of this scheme is that hardware protection between objects that reside on the same workstation is not provided. Consequently, an object failure can be potentially fatal to an entire workstation.

2.3.7 TABS / Camelot

TABS and Camelot [Eppinger and Spector 1985; Spector et al. 1985; Spector et al. 1986; Spector 1987; Spector et al. 1987] support large-grain and medium-grain objects and the static variation of the active object model. They differ from most DOBPSs in that a collection of medium-grain objects is encapsulated by a number of server processes. Each server process accepts invocation requests that specify the medium-grain object and the operation to be invoked.

See Figure A2 for a summary of the object structure schemes supported by each system.

3. OBJECT MANAGEMENT

Objects are the fundamental resources of a DOBPS; therefore, their management is an essential function of these systems. This section describes the features provided by a DOBPS for making the effects of an action on persistent objects

permanent, for synchronizing the execution of multiple concurrent invocations within an object, for protecting objects from unauthorized clients, and for recovering objects that fail. Since synchronization, security, and recovery mechanisms used in DOBPSs are similar to those used in other systems [Svobodova 1984; Tanenbaum and Renesse 1985], only an outline of the more common techniques is given here. A brief overview of how objects are managed in a number of existing systems is then presented.

3.1 Action Management

An important function of a DOBPS is to manage the activities of its actions. Actions should have the following three properties:

- *Serializability*. Multiple actions that execute concurrently should be scheduled in such a way that the overall effect is as if they were executed sequentially in some order.
- *Atomicity*. An action either successfully completes or has no effect.
- *Permanence*. The effects of an action that successfully completes is not lost, except in the event of a catastrophic failure.

A single action may produce a number of related invocations that may affect multiple objects. Typically, an action successfully completes only when all the invocations associated with it complete. An action that successfully completes is said to *commit*, whereas an action that fails to do so is said to *abort*. If an action commits, all modifications made to persistent objects by the action are made permanent by recording the changes to secondary storage (see Section 5.1.2). If an action aborts, all modifications it made to objects are undone.

The commit procedure ensures that either all the modifications made to objects by an action are made permanent or none are. The most popular commit procedure is the two-phase commit protocol [Gray 1978]:

(1) A *precommit* request is issued to all the affected objects.
(2) An object that receives a precommit request writes its modifications to secondary storage and returns an acknowledgment.
(3) If all the objects return an acknowledgment, a *commit* request is issued to the affected objects. If any object fails to respond to its precommit request within some time frame, however, an *abort* request is issued to all the affected objects, and the commit procedure is terminated.
(4) An object that receives a commit request makes the changes permanent, resets the control structures associated with the action (returning obtained locks, for example), and returns a second acknowledgment. An object that receives an abort request discards the changes and resets the control structures.
(5) A commit request is repeatedly issued to the objects that fail to respond until all return acknowledgments, at which point the commit procedure ends.

This commit protocol is simple and efficient, but one problem with the scheme is that it may block if the object coordinating the protocol fails or if a network failure occurs.

Initiating the procedure to commit an action may be the responsibility of either the DOBPS or its users. The simplest scheme is the *request* scheme. In this scheme, the user is responsible for deciding when to initiate the commit procedure. A user is not required to commit each and every action. This can improve the performance and the response time of a DOBPS, since executing the commit procedure can be expensive. The main problem with not committing every action is that the guarantee of permanence and atomicity are lost. For example, one action can modify a number of persistent objects and complete without committing. A second, subsequent, action can modify some, but not all, of these objects and then commit. If the system fails and in the in-memory versions of these objects are lost, some of the effects of the first action will be lost, thereby placing

the system in an invalid or inconsistent state when the system is restored (see Section 3.4).

To guarantee that all the properties of an action will be enforced, the system must be responsible for managing the actions and committing them when they are complete. In the *transition* scheme [Gray 1980], when all the invocations associated with an action successfully complete, the system performs the commit procedure to make the modifications permanent. Successful completion of all invocations is essential to the outcome of a transaction; the failure of any invocation causes the corresponding transaction to abort. This is a simple, straightforward scheme that ensures that either all the modifications made by an action are completed and committed or none are. Two drawbacks of this scheme are the overhead of the commit procedure occurs whenever an action completes, and an action is forced to abort if any of its invocations fail.

An extension of the transaction scheme is the *nested transaction* scheme [Moss 1985]. In this scheme, a single *top-level action* creates and manages multiple lower level, autonomous *subactions*. The failure of a subaction does not necessarily force the top-level action to fail as well. A top-level action can handle a subaction failure in any way it chooses. For example, it may choose to perform the subaction again, it may abort the entire action, or it may simply ignore the problem. The modifications made to objects by subactions that successfully complete are conditional to the success of their top-level action. Only when a top-level action commits are the modifications created by its subactions made permanent. If a top-level action aborts, all the changes made by its subactions are also undone. The nested transaction scheme permits a finer degree of control over an action by enabling failures to be contained and handled while allowing progress to be made elsewhere. One disadvantage of the nested transaction scheme is that the representation of objects in memory is more complex in order for the system to be able to undo the effects of a failed subaction (see Section 5.1.1).

3.2 Synchronization

Another important function of a DOBPS is to ensure that the activities of multiple actions that invoke the same object do not conflict or interfere with one another. An action should not be permitted to observe or modify the state of an object that has been partially modified by another action that has not committed. Failing to ensure this can lead to a condition known as *cascading aborts*. That is, any action that observes the partially modified state of an object resulting from an action that later aborts also has to be aborted. To ensure that all actions have the property of serializability and to protect the integrity of the objects' states, a synchronization mechanism is required. Many synchronization schemes exist, most of them classified as being either pessimistic or optimistic schemes [Bernstein and Goodman 1981].

In a *pessimistic* synchronization scheme, the system takes appropriate steps to prevent conflicts from occurring. An action that invokes an object is temporarily suspended if it will interfere with another action that is currently being serviced by the object. When all conflicting actions commit, the suspended action is resumed. Read/write locks are the most common mechanisms used by a pessimistic synchronization scheme; however, timestamps, semaphores, and monitors are also used.

In an *optimistic* synchronization scheme [Mullender and Tanenbaum 1985], an object does not take steps to prevent conflicts from occurring while invocations are being processed. Instead, before an action can commit, it is tested for serializability to ensure that the information it has observed does not conflict with changes made by another action that has previously committed. That is, the system determines if the data that were examined by an action in its version of the object is still up to date or whether the data have been altered by an action that has since committed. If no

data have changed, the action can commit. If the action examined some information that is now out of date, however, its modifications based on that information will be incorrect and consequently the action must be aborted. To reduce the likelihood of conflicts, objects may be represented either as a number of pages or as a number of medium-grain and/or fine-grain objects that can be modified independently. The optimistic synchronization scheme permits the maximum degree of concurrency possible within an object; actions are never suspended, since they are in a pessimistic synchronization scheme.

The major problem with the optimistic scheme is that some actions that successfully complete may still be forced to abort. Furthermore, multiple copies of each object must be maintained in memory to permit concurrency and the changes made by each committed action must be recorded in secondary storage to enable the commit procedure to test for serializability (see Section 5.1).

A pessimistic scheme avoids the overhead of undoing and redoing requests at the expense of reduced concurrency. For example, two actions that examine and modify different parts of the same object are not able to execute concurrently, even when there is no problem of conflict. An optimistic scheme, on the other hand, avoids the overhead of delaying requests at the expense of undoing and redoing requests. Therefore, a pessimistic scheme performs better than an optimistic scheme when conflicts are frequent, whereas the reverse holds true when conflicts are infrequent.

3.3 Security

Providing a security scheme to prevent unauthorized clients from successfully invoking an object is an important, but often ignored, function of a DOBPS. This is especially important in a multiuser system, where users are assigned different security clearances and are permitted to operate on different sets of objects. The security mechanism may be provided either by the system or by the user.

A common security mechanism is the *capability* scheme that incorporates protection into the naming scheme [Cohen and Jefferson 1975; Tanenbaum et al. 1986]. A capability is a key consisting of two fields: a name field and an access rights field. The name field specifies a particular object; the access rights field indicates the specific operations of the object that may be invoked. Each capability has exactly one object; however, the same object may have multiple capabilities. This enables the owner of an object to vary the access rights for different clients. Each server object passes its capabilities among the clients of the system that request it. In order for a client to make an invocation of an object, it must first possess one of the object's capabilities. This capability is passed as a parameter in the invocation request. Either the system or the objects can be responsible for verifying, managing, and maintaining the capabilities. In either case, capabilities must be protected to ensure they are not modified or forged.

Another type of security mechanism is the *control procedure* scheme [Banino and Fabre 1982]. In this scheme, every object has a special procedure through which all incoming invocation requests must first pass. These procedures check the authorization of clients making a request and terminate all invalid requests. The control procedure scheme is flexible in that it can support almost any type of security scheme. For example, it can use a mechanism that ensures that a request is not received from a client that has not been previously placed on a list of authorized clients. Or it can use a password scheme to block unauthorized requests. This permits each object to provide security tailored to the requirements of the object. If an object is unimportant, a minimal security scheme can be used; if an object is confidential, multiple schemes can be used. Bypassing the security mechanism is extremely difficult because each object is entirely responsible for enforcing its own protection scheme.

3.4 Object Reliability

A DOBPS must be able to detect and recover from object and workstation failures.[5] This is an important feature because as the number of components in a system increases, the probability that a failure will occur also increases. There are two general methods of providing object reliability. One method is to *recover* a failed object as quickly as possible, limiting the amount of time it remains unavailable. An alternative method is to *replicate* objects at multiple workstations so that each object has a high probability of surviving a workstation failure. Object recovery techniques include roll-back and roll-forward recovery schemes. Replication schemes include primary copy and peer object schemes.

3.4.1 Object Recovery

In a *roll-back* recovery scheme, a failed object is restored to its last consistent state that was recorded in secondary storage by the commit procedure. All processes and invocations that were in progress when the failure occurred are lost. Roll-back recovery schemes are simple and efficient. Another important property of these schemes is that they have a high probability of successful recovery.

In a *roll-forward* recovery scheme [Powell and Presotto 1983], a failed object and all objects with which it was working are restored to their last consistent states that were recorded by the commit procedure. All processes and invocations that were in progress at the time of the failure are then restarted and allowed to complete. The roll-forward recovery scheme is more complex than a roll-back scheme because the system makes it appear that the failure did not occur. Due to the many ways in which an object may fail and the complex interactions between objects, however, there is no guarantee that recovery will be successful. For example, if the mechanism fails to record the intentions of all the actions of the system correctly, the entire recovery procedure may fail. Another problem associated with roll-forward recovery schemes is an object failure caused by a software error will occur again when the object is restored. This may cause recovery to take place again, the failure to occur again, and so on. The topic of object recovery is further discussed in Section 5.1.2.

3.4.2 Object Replication

A replication scheme permits copies of an object to exist on multiple workstations. This enables a DOBPS to tolerate a number of workstation failures while still allowing it to provide full functionality. The failure of any workstation only results in the unavailability of replicas that reside on that workstation; a replica that resides on another workstation should be able to continue servicing invocation requests. There are a number of problems associated with replicating objects, including maintaining consistent information between replicas and synchronizing the activities of multiple clients. In addition, a network partition can create havoc with some replication schemes.

The simplest scheme is to allow only *immutable objects* to be replicated. Immutable objects can only be examined by clients; their states cannot be modified. These types of objects can be replicated without the problems of maintaining state consistency and synchronizing access. A deficiency of this scheme is that it is of limited use since only nonmodifiable objects may be replicated.

An alternate approach is the *primary copy* replication scheme [Alsberg and Day 1976]. In this scheme, one object replica is designated as the primary copy, whereas the other replicas are ordered and maintained on separate workstations as secondary copies. Nonmodifying, or read, requests can be handled by any replica. Modifying, or write, requests

[5] In most systems, network failures are indistinguishable from machine failures. Therefore, for the purposes of this paper, they will not be considered separately.

must be serviced by the primary copy, which then propagates the modifications to each of the secondary copies. There are two variations of the primary copy scheme: a static primary copy and a dynamic primary copy. In the *static* primary copy scheme, if a primary copy fails, the corresponding object is prevented from servicing modifying requests until the primary copy is recovered. This scheme provides additional object availability for read requests but not for write requests. The *dynamic* primary copy scheme, on the other hand, provides object availability for write requests. If a primary copy fails, the system assigns one of its secondary copies to take over as the new primary.

There are additional problems associated with the dynamic primary copy replication scheme. First, when a secondary copy replaces a primary copy it must take over the identity and the resources such as ports or locks of the failed object so its clients do not perceive any change in the server object. Second, a network partition that separates a primary copy from some of its secondary copies may result in two sets of objects being created, with a primary copy on either side of the partition. When the partition is repaired, conflict resolution may have to take place to remove any discrepancies.

The *peer objects* replication scheme is a third approach. In this scheme there is no designated primary object or secondary objects; instead, every replica is considered to be equal. Both modifying and nonmodifying requests can be serviced by any replica; however, the cooperation of some or all of the other replicas are required in order to process the request. These schemes permit the failure of a limited number of replicas to be tolerated without preventing write requests from being processed. They are also less problematic when network partitions occur. There are a number of variations of the peer objects scheme, including Voting [Gifford 1979], Available Copies [Bernstein and Goodman 1984], and Regeneration [Pu et al. 1986].

3.5 Overview of Object Management in Existing Systems

3.5.1 Amoeba

Amoeba supports the request scheme, as well as both a pessimistic and an optimistic synchronization mechanism. The pessimistic locking scheme is used when an action affects multiple objects, whereas the optimistic scheme is used when only a single object is affected. In the optimistic scheme, an action is given a copy of the most recent version of the object on which to make its modifications. To reduce the chance of conflict, an object's state is divided into pages. Two tests are used to check for serializability. The first test determines if the version of the object from which the modified version was derived (the past version) is still the most recent version (the current version). If it is, the commit procedure is initiated. Otherwise, the pages of the current version that were examined by the action are compared against the corresponding ones in the past version. If none of the pages has changed, the action did not observe some information that has changed. A new version of the object is created by combining the updates made to create the current version and the modified version, and the commit procedure is initiated. If both tests fail, the action is aborted.

Security is provided by a capability scheme that uses a random number generator that produces large, sparse numbers as object identifiers [Mullender and Tanenbaum 1986]. This scheme makes it difficult, though not impossible, for a user to forge a capability. Further protection can be provided by encrypting capabilities or by using special hardware to ensure that only the server object, which is specified in the invocation, is the one that actually receives the request.

A special boot server object is used to detect and recover objects and workstations that have failed. An object can register with the boot server to ensure that it is not down for a lengthy period of time. Periodically, the boot server probes

each of its registered objects to determine if they are still functioning. Any object that does not respond to a number of probes is assumed to have failed, and the recovery procedure is performed. If the object has failed but the workstation on which it was residing is still working, the object is restored on that workstation. If the workstation has failed, it is restarted and the object is restored. A roll-back recovery scheme that uses checkpoints is supported, but an object replication scheme is not provided.

3.5.2 Argus

Argus supports the nested transaction scheme. Each large-grain object visited by an action records which of its medium-grain objects were examined and which were modified. When an invocation completes, the identifier of the large-grain object that was servicing the request is added to a list of all large-grain objects that were affected by this action. This list is eventually propagated back to the object that initiated the action. This object then performs the commit procedure on all of the large-grain objects it affected. Each large-grain object, however, is responsible for performing the commit procedure on each of its affected medium-grain objects.

A pessimistic synchronization scheme that uses read/write locks is supported. The usual locking rules are simplified by not permitting a top-level action to execute concurrently with its subactions. This ensures that only one action or subaction can modify an object at any one time. When a subaction successfully completes, its locks are inherited by its parent action.

Each workstation provides a guardian manager that is responsible for creating new large-grain objects at the workstation, for detecting objects that have failed, and for recovering objects after a failure. A partial roll-forward recovery scheme that uses a commit log is supported (see Section 5.1.2). Argus does not provide a security scheme or an object replication scheme.

3.5.3 CHORUS

CHORUS supports the request scheme. A single action may create multiple subactivities that execute concurrently. This scheme is similar to the nested transaction scheme, but it is not as flexible or as powerful. In CHORUS, a subactivity is a completely separate entity that may commit independent of the top-level action.

A pessimistic synchronization scheme is supported since each object can perform at most one invocation request at a time. The control procedure security scheme and a roll-back recovery scheme that uses checkpoints are provided.

CHORUS uses a variation of the primary copy replication scheme. A single secondary copy acts solely as a backup and is otherwise not used. The primary copy and the backup copy consecutively perform the same invocations. The primary copy accepts and services invocation requests and propagates completed requests to the backup copy. The backup copy services the invocation requests delivered to it but suppresses the reply it would normally issue when it completes. As a result, the backup copy performs exactly the same operations as the primary copy, but delayed by one operation so it records the last consistent state of the primary copy. The primary copy is responsible for periodically checking the status of the backup copy and vice versa. If the primary copy fails, the backup copy takes its place and creates a new backup object. If the backup copy fails, the primary copy creates a new backup copy to take its place.

Each workstation provides an inspector object that periodically checks with its counterparts in the network to detect workstation failures. If a failure is detected, the appropriate roll-back recovery procedure is performed.

3.5.4 Clouds

Clouds supports the nested transaction scheme. An action manager object is responsible for maintaining a record of all

objects visited by an action and for coordinating the commit procedure.

Two pessimistic synchronization schemes are provided: an automatic scheme and a custom scheme. The automatic scheme uses read/write locks with each operation of an object defined as requiring either a read or a write lock. The custom scheme permits a user to create specific synchronization rules using either semaphores or locks. Custom synchronization has the benefit of enhancing concurrency because it permits semantic knowledge about the operations to be used. The drawback of custom synchronization is that the serializability property of actions cannot be enforced by the system since users define their own rules. A capability scheme is provided by Clouds for security.

Clouds supports the roll-back recovery scheme that uses checkpoints and the peer objects scheme for object replication. It also permits immutable objects to be replicated.

3.5.5 Eden

Eden supports a variation of the nested transaction scheme in which an action must be explicitly committed by the user. To manage the activities of each nested transaction, a special manager object is created. Each manager object is responsible for interacting with the objects affected by the corresponding action and for coordinating the commit procedure when the client informs it to do so.

A pessimistic synchronization scheme that uses monitors, a capability security scheme, and a roll-back recovery scheme that uses checkpoints are provided.

Object replication is supported using the regeneration version of the peer objects scheme [Pu et al. 1986]. For each replica of an object, a version manager (see Section 5.3) is created at the workstation on which the replica resides. The version managers interact to ensure that inconsistencies do not arise among the replicas. Eden also permits immutable objects to be replicated without restriction.

3.5.6 Emerald

Emerald supports the request scheme, a pessimistic synchronization scheme that uses monitors, and a roll-back recovery scheme that uses checkpoints. A security scheme is not provided. Emerald only permits immutable objects to be replicated.

3.5.7 TABS / Camelot

TABS supports the nested transaction scheme. Each workstation provides a Transaction Manager that is responsible for handling the commit procedure on all affected objects that reside on the workstation. Each Transaction Manager is also responsible for acting as the coordinator for the managers of those workstations that are its children, with respect to the committing action. Consequently, a number of managers must cooperate in order to commit an action that spans multiple workstations. If an action aborts, the Transaction Managers ensure that the partial effects of the action are undone.

Camelot also supports the nested transaction scheme. Two commit procedures are provided: a *blocking* commit based on the two-phase commit protocol and a *nonblocking* commit that is a combination of the three-phase and the Byzantine commit protocols [Spector et al. 1987]. The nonblocking commit procedure has a higher overhead than the blocking commit procedure; however, the likelihood that an object will remain locked due to a failure during the commit procedure is reduced.

Both TABS and Camelot support a pessimistic synchronization scheme. A *type-specific locking* scheme [Korth 1983] is provided to permit a programmer to define customized lock modes and protocols so they can be tailored to the requirements of the objects. This scheme is similar to the one supplied by Clouds. Neither a security scheme nor an object replication scheme is provided.

Individual objects cannot be recovered; instead, an entire workstation must be recovered when a failure occurs. A

roll-back recovery scheme that uses an undo/redo log is supported (see Section 5.1.2). During the recovery procedure, the Recovery Manager of the failed workstation determines whether the invocations need to be redone or undone and issues the appropriate requests to the affected objects. See Figure A3 for a summary of the Object Management schemes supported by each system.

4. OBJECT INTERACTION MANAGEMENT

A DOBPS is responsible for managing the invocations between cooperating objects. When an action makes an invocation request, the system must locate the specified object, take the appropriate steps to invoke the specified operation, then possibly return a result. This section describes the features provided by a DOBPS for locating server objects, for handling object interactions, and for detecting invocation failures. It then presents a brief overview of how object interactions are managed by a number of existing systems.

4.1 Locating an Object

A DOBPS should provide the property of *location transparency* so a client does not have to be aware of the physical location of an object in order to invoke it. Whenever an invocation is made, the system must determine which object was invoked and on which workstation the object currently resides in order to deliver the request to it. As was mentioned previously, the system must assign an identifier to each object. These identifiers must be unique; they should not change during their lifetime and once used should not be reused. The mechanism for locating an object should be flexible enough to allow objects to migrate or move from one workstation to another.

One scheme is to *encode* the location of an object within its object identifier. When an invocation is made, the system simply examines the appropriate field of the specified object identifier in order to determine the workstation on which the object resides. This is a straightforward and efficient scheme. One restriction of the scheme, however, is that an object is not permitted to move once it is assigned to a workstation, since this would require its identifier to change. Consequently, an object is fixed to one particular workstation throughout its lifetime.

A second approach is the *distributed name server* scheme. In this scheme, the system creates a group of name server objects that are maintained on a number, but not necessarily all, of the workstations. These objects cooperate with one another so that collectively they contain up-to-date information about the location of every object in the system. There are two variations of this scheme. In the first variation, a name server maintains a complete collection of location information so each server can service any location request. In the second variation, partial information can be maintained by each server; if a location request cannot be serviced by one server, it is delegated to another. The major problem with this scheme is that at least one server must be notified every time a new object is created or an object is moved from one location to another. Information maintained by these servers may be slightly out of synch since updating a name server is not an instantaneous operation and maintaining consistent information among the multiple components of the name server can be difficult.

Another approach is the *cache/broadcast* scheme. A small cache is maintained on each workstation that records the last known locations of a number of recently referenced remote objects. When a client makes a remote invocation, the cache is examined to determine if it has an entry for the invoked object. If a location is found, the invocation request is sent to that workstation. If the object no longer resides at that workstation, however, the request is returned. If the location of the object is not recorded in the cache or the cache information is found to be outdated, a message is broadcast throughout the network requesting the current location of the object. Every workstation that receives a broadcast

request does an internal search for the specified object. If the object is found, a reply message is returned to the workstation that made the request and its cache is updated.

The cache/broadcast scheme can be very efficient, since an object's location may be found in the local cache. It is also flexible, since it permits an object to be moved from one workstation to another while avoiding the expense and delay of having to notify other workstations or a distributed name server. One problem with this scheme, however, is that broadcast requests will clutter up the network, disturbing all the workstations even though only a single workstation is directly involved with each location request.

Forward location pointers can be used to enhance most location schemes. A forward location pointer is a reference used to indicate the new location of an object. Whenever an object is moved from one workstation to another, a forward location pointer is left at the original workstation. To locate an object that has been moved, the system can simply follow the forward pointer or chain of pointers to the workstation on which the object currently resides. One problem with using forward location pointers is that they introduce additional system overhead for upkeep. Additionally, this scheme cannot completely handle the problem of finding migrating objects since some pointers may be lost or may be unavailable due to workstation failures.

4.2 SYSTEM-LEVEL INVOCATION HANDLING

When a client makes an invocation on an object, the DOBPS is responsible for performing the necessary steps to deliver the request to the specified server object and for returning a result back to the client. How a system handles invocations depends entirely on the object model supported. Two schemes that are used are the *message passing scheme* and the *direct invocation scheme.*

4.2.1 Message Passing

A DOBPS that provides the active object model typically supports the pure *message passing* scheme to handle object interactions. When a client makes an invocation on an object, the parameters of the invocation are packaged into a request message. This message is then sent to a server process or a port associated with the invoked object. A server process in the invoked object accepts the message, unpacks the parameters, and performs the specified operation. When the operation completes, the result is packaged into a reply message, which is then sent back to the client.

DOBPSs differ from most distributed systems in that two interacting processes do not go through the effort and expense of setting up and tearing down a static, heavyweight connection. Instead, the binding of a client and a server is lightweight and done dynamically on every invocation. This approach is more suitable for the request/response communication pattern common to these systems. It is also better suited for the multiple machine environment of a DOBPS because it maps more naturally onto the mechanisms required for intermachine communication. This permits a DOBPS to more easily support features such as object mobility (see Section 5.2). A drawback of the message passing scheme is the overhead of message passing for intramachine invocations.

4.2.2 Direct Invocation

A DOBPS that provides the passive object model typically support the *direct invocation* scheme to handle object interaction. In the passive object model, a single process is responsible for performing all the operations associated with an action. As a result, a process will migrate from operation to operation and from object to object whenever the corresponding action makes an invocation.

When a process invokes a server object that resides on the same workstation, the following four steps are taken (Figure 7):

(1) The state of the process and the objects in which it currently resides are recorded in the stack space of the process. The system may protect the stack to ensure that this information

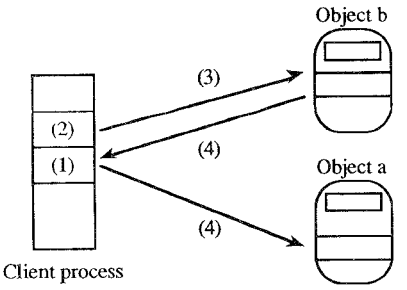


Figure 7. Direct invocation, local request.

cannot be examined or corrupted by the activities of subsequent invocations.

(2) The parameters of the invocation are added to the stack.

(3) The invoked object is loaded into memory, and a procedure call is made to start the process executing the appropriate code.

(4) When the operation terminates, the results are returned to the client and the process is restored to the state it was in before the invocation.

When a process invokes a server object that resides on a different workstation, the following three steps must also be performed, the first two after step (1) and the third after step (3) (Figure 8):

(1.1) A message containing the parameters of the invocation is created and sent to the workstation on which the server object resides.

(1.2) The workstation that receives this message creates a worker process to execute on behalf of the original process.

(3.1) When the operation terminates, a message containing the results of the invocation is created and returned to the workstation on which the original process resides. The worker process is then killed.

An invocation on a local object is similar to a procedure call, whereas an invocation on a remote object is similar to a remote procedure call.

The direct invocation scheme should incur less performance overhead than the message passing scheme when it comes to local invocations since interactions between objects that reside on the same workstation are relatively efficient. When it comes to remote invocations, on the other hand, the direct invocation scheme has the added expense of creating and destroying worker processes.

4.3 Detecting Invocation Failures

An invocation failure can be classified as being either an *existing fault* or a *transient fault*. An existing fault is defined as a failure that occurs before an invocation is started. The most common type of existing fault occurs when an invoked object cannot be located. These types of faults are relatively easy to detect and handle.

A transient fault, on the other hand, is a failure that occurs while an invocation is being performed. Transient faults are failures that occur sometime after a server object has accepted an invocation request but before the modifications made to it have been made permanent by the successful completion of the commit procedure. These faults are much more difficult to detect and handle because there are many different ways an invocation can fail. For example, the failure of an invocation may be caused by the failure of the client object, the failure of the server object, or a network partition that separates a client from its server. A DOBPS should provide mechanisms for both client and server objects of the system to detect and recover from transient failures. Numerous failure detection schemes exist, including ones that use time outs, object probes, and invocation probes [Liskov et al. 1987; Tanenbaum and van Renesse, 1987]. Several such schemes are outlined in Section 4.4.

If the failure of an invocation is not detected by the client object, the client and the corresponding action may block and wait indefinitely. Consequently, a client must be able to detect invocation failures and initiate a recovery procedure when one occurs. A recovery procedure

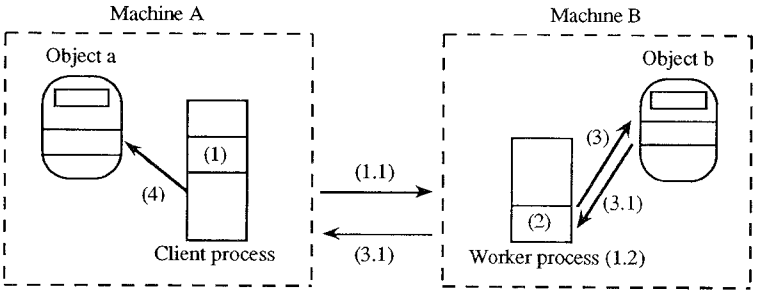


Figure 8. Direct invocation, remote request

typically releases the resources held by the client and notifies the corresponding action of the failure. It may also attempt to reissue the invocation request at a later time.

If the failure of an invocation is not detected by the server object, valuable system resources may be tied up unnecessarily. An object that starts an invocation but its results are no longer wanted is referred to as an *orphan*. An orphan may arise due to the failure of a client object, an aborted transaction, a workstation failure, or a network partition. Orphans waste system resources since they may hold locks, thus causing a loss of throughput. Consequently, they should be eliminated as quickly as possible.

4.4 Overview of Object Interaction Management in Existing Systems

4.4.1 Amoeba

Amoeba supports the message passing scheme that uses ports. The cache/broadcast scheme is used to locate the ports associated with an object. An object probe scheme is used by both the clients and the servers of the system to detect invocation failures. When a server object receives a request, its receipt is acknowledged so the client knows it has arrived safely. If the client fails to receive a result a certain period of time after the acknowledgement, it sends an "Are you alive?" message to the server object, which responds as soon as it is able. If the client does not receive a reply to this message, it concludes that the server and hence the invocation have failed. Similarly, if the server stops receiving "Are you alive?" messages from a client, it concludes the client has failed and the corresponding orphan invocation is killed.

4.4.2 Argus

Argus encodes the locations of objects within their identifiers. The message passing scheme is supported. To aid in the mapping of reply messages to the appropriate client processes, each process is assigned a unique identifier in the large-grain object in which it resides. This identifier is passed in the invocation message and returned in the reply message to identify the process that made the invocation.

An invocation probe scheme is used by client objects to detect invocation failures. If a client fails to receive a result after a certain period of time, it sends a probe message to a special process of the large-grain object that was invoked. An object that receives a probe message determines if a process is currently executing in the object on behalf of the sender of the probe. If there is, an acknowledgment that the invocation is still being executed is returned; if not, a negative acknowledgment is returned. A client that receives a negative acknowledgment or fails to receive an acknowledgment concludes that its invocation has failed.

A status report scheme [Walker 1984] is used by servers to detect invoca-

tion failures. Each large-grain object maintains (in stable storage) three data structures that are used by the orphan detection scheme: a crash counter, a done list, and a map list. The crash counter indicates how many times the object has failed. The done list records all actions that are known by the object to have failed. The map list records the latest known crash counts of all large-grain objects known by the object. Periodically, the done list and the map list of a large-grain object are piggybacked onto an invocation request message. A large-grain object that receives this information checks its server processes against the done list to determine if any of them are descendents of an aborted action. It then compares its map list against the one received to determine if any of its current clients resided on a workstation that has failed. All orphans are then destroyed. Finally, the done and map lists of the large-grain object that received this information are updated to reflect the changes in the system.

4.4.3 CHORUS

CHORUS supports a message passing scheme that uses ports. Intermachine communication is done by passing messages to a surrogate local port that is managed by a network server. It is the responsibility of the network server to handle the transportation of messages over the machine boundaries in a transparent fashion.

A combination of the encoding scheme and the cache/broadcast scheme is used to locate the ports of an object. Encoded into each port identifier is the identity of the workstation on which the port was originally created. In addition, the system maintains on each workstation a record of the current location of each port that was originally created on the workstation. When the location of a port is required, the port identifier is examined to determine the workstation that maintains the information and a location request is sent to it. A workstation that receives a location request returns a result containing the location information as soon as it is able. If the location of a port cannot be obtained because the corresponding workstation could not be contracted or the location returned is found to be incorrect, a location request is broadcast throughout the network. A simple time-out scheme is used by clients to detect invocation failures.

4.4.4 Clouds

Clouds supports the direct invocation scheme. When an invocation is made on an object that does not reside locally, the local communication manager object creates an invocation message. This message contains the capability of the object being invoked, the name of the operation being invoked, and a copy of the parameters. The communication manager broadcasts this message to its counterparts in the network as a "search and invoke," request and the client is suspended. A communication manager that receives a "search and invoke" request determines if the object resides on its workstation. If the object is not found, the request is simply ignored. If the object is found, the communication manager accepts the request and creates a worker process to execute on behalf of the client process. The worker process copies the parameters of the invocation onto its stack and invokes the specified operation. When the operation terminates, a reply message is constructed and sent back to the communication manager of the client's workstation. The client process is then unblocked and the results are passed to it.

The search procedure is made more efficient by the use of three structures: an active object table, a maybe table, and an object directory [Pitts and Dasgupta 1988]. The active object table records all active objects that reside on the workstation. The maybe table records that either the object may reside locally or that it definitely does not. The object directory records all active and inactive objects that reside on the workstation. The search procedure examines each structure in turn when it attempts to locate an object.

This enables a number of quick and efficient membership tests to be performed before resorting to the task of examining the entire contents of the workstation. A simple object probe scheme is used by the clients to detect invocation failures.

A time-out scheme is used by the servers to detect invocation failures and orphans [McKendry and Herlihy 1985]. When an action acquires a lock at a workstation, it is assigned two times: a quiesce time and a release time. The quiesce time indicates when the action may no longer execute operations at that workstation, although it may still commit or abort. The release time indicates when the invocation is concluded to be an orphan. If the status of an action is unknown when its release time arrives, the invocation is terminated, all its modifications are discarded, and all locks held by it are released. To alleviate the problem of an action aborting unnecessarily, a refresh protocol is used to advance periodically the quiesce and release times of each action that is still functioning properly.

4.4.5 Eden

Eden supports the message passing scheme and a pure cache scheme for locating objects. A simple time-out scheme is used by client objects to detect invocation failures.

4.4.6 Emerald

Emerald provides a cache/broadcast scheme that uses forward pointers to locate objects. Each cache entry is coupled with a timestamp to indicate the relative age of the data. When an invocation is made, the local cache is examined. If a location is found in the cache but the information is found to be out of date, the specified workstation is queried to determine if it knows of a more recent location for the object. If it does, that location is checked. A broadcast location request is issued if the location of the object cannot be determined. Emerald ensures that each workstation of the network receives this request and that all available workstations are searched.

Emerald supports the direct invocation scheme. Whenever a process makes an invocation on an object, an activation record is created for the process. If the invocation is made on an object that resides locally, the activation record is created on the top of the process' current stack. If it is made on an object that resides on a remote workstation, the activation record forms the base of the new process' stack on that workstation. A simple time-out scheme is used by client objects to detect invocation failures.

4.4.7 TABS / Camelot

Both TABS and Camelot use a pure broadcast scheme to locate objects. The system maintains on each workstation a list of all objects that reside on it. When an invocation is made on an object that does not reside locally, a location request is broadcast to all the workstations in the network. The workstation on which the object resides returns an acknowledgment.

A message passing scheme that uses ports is supported. Transparent intermachine communication between a client and a server is provided by a pair of communication managers. The communication managers supply two ports: one that resides on the workstation of the client and one that resides on the workstation of the server. Both the client and the server send their messages to their local port while the communication managers handle the mapping and transportation of the messages to the corresponding port. A simple time-out scheme is used by the clients to detect invocation failures.

See Figure A4 for a summary of the Object Interaction Management schemes supported by each system.

5. RESOURCE MANAGEMENT

A DOBPS like any other distributed operating system must provide mechanisms to manage the physical resources

of the system, including primary memory, secondary storage devices, processors, and workstations of the network. Specific to DOBPSs are how objects are represented in memory and in secondary storage, how they are transferred between these two resources, and how they are assigned to processors. This section outlines those aspects of resource management related to objects. It also presents a brief overview of the way in which resource management is handled in a number of existing systems.

5.1 Memory and Secondary Storage

Objects that are lost if the workstation on which they reside fails are said to be *volatile*. Volatile objects are temporary, reside solely in memory, and are relatively inexpensive to maintain and use. Objects that can survive the failure of their workstation with a significantly high probability are said to be *persistent*. A persistent object resides in secondary storage; however, one or more working copies of this object may reside in memory. Actions modify a persistent object's version in memory. The version in secondary storage is typically updated only when an action commits. This ensures that a stable, consistent version of each object is maintained at all times. Persistent objects are more expensive to maintain and use than volatile objects because of these additional overheads.

A persistent object that resides both in memory and in secondary storage is said to be *active*. A persistent object that is maintained solely in secondary storage device is said to be *inactive*. When an invocation is made on an inactive object, a volatile copy of the object is created and loaded into memory to make the object active. New processes are created for the volatile version if necessary. When the commit procedure is successfully performed on an active object, the object is copied back to secondary storage and may be deactivated so the system can reclaim the memory it occupied. This automatic loading and unloading of objects into and out of memory is performed transparently by a DOBPS to hide the fact that a secondary storage device is used and to give the system the appearance that objects are always available to be invoked.

5.1.1 Representation of Objects in Memory

The way in which an object is represented in memory depends both on the synchronization scheme and the action management scheme supported by the DOBPS. The synchronization scheme influences the number of versions of each object that are maintained, whereas the action management scheme influences the representation of each version.

When a DOBPS supports a pessimistic synchronization scheme, typically a *single version* of each object is maintained in memory. In this scheme, all actions that invoke an object modify the same volatile version. When an optimistic synchronization scheme is supported, *multiple versions* of each object are created and maintained in memory. In this scheme, every action that invokes an object is assigned its own volatile version of the object on which to perform its modifications. This enables multiple actions to invoke the same object simultaneously while ensuring that they will not interfere with one another.

The representation of each object in memory depends on whether or not the nested transaction scheme is supported. In a DOBPS that supports the request or transaction scheme, the traditional approach of representing a version in memory as an exact copy of the corresponding object is sufficient. If an action fails, the volatile version can simply be discarded. When the nested transaction scheme is supported, this approach is usually not adequate since multiple subactions of an action can modify the same object. The problem is that the changes made to the object are not made permanent until the top-level action commits and each subaction can complete or fail independent of the others. If a subaction modifies an object and is then forced to abort, the changes it made must be undone so that the object is restored to its state before

the execution of the subaction. Consequently, an additional mechanism is needed to undo the changes made by a failed subaction. These schemes include using an undo/redo log (see Section 5.1.2) or maintaining an immutable *hot standby* object that records the state of the version that was created by the last subaction to complete successfully.

5.1.2 Representation of Objects in Secondary Storage

DOBPSs must record enough information in secondary storage so that a persistent object can be restored to a consistent state should it or the workstation on which it resides fail. A DOBPS may record the entire state of an object whenever it is committed (checkpoint schemes), or it may simply record the relative changes made to the object since some previously recorded state (log schemes).

Checkpoint Schemes. In the *checkpoint* scheme, the entire state of a modified object is recorded onto secondary storage when the corresponding action commits. During the precommit stage of the commit procedure, the modified version of an object is written to secondary storage without disturbing the old version. During the commit stage, the modified version replaces the old version, which is then discarded. If at any time the action aborts, the modified version is discarded and the old version remains unaffected.

One advantage of the pure checkpoint scheme is that it makes efficient use of secondary storage, since only a single copy of each object is maintained. A problem with this scheme is its relatively large performance overhead, since the entire state of a modified object is recorded whenever an action is committed. This is especially true if only a small change is made to an object that has a very large state. The pure checkpoint scheme also has the drawback that only the most recent checkpoint is maintained; older checkpoints are not available. This scheme does not record enough information to support an optimistic concurrency control scheme sufficiently, which requires the recent history of an object be maintained so that the serializability test can be performed (see Section 3.2).

A variation of the checkpoint scheme is the *history of checkpoints* scheme. In this scheme a persistent object is represented in secondary storage as an ordered collection of checkpoints. Instead of destroying the old version of an object when a new version is checkpointed, the new version is added to the sequence of checkpoints. The history of checkpoints scheme enables the previous checkpoints of an object to be examined. Two disadvantages of this scheme are the additional storage space used to record the extra checkpoints and the introduction of the problem of determining when old checkpoints are no longer needed and therefore should be deleted.

Log Schemes. In a *log* scheme, whenever a persistent object is modified, the relative changes made are recorded in a common log maintained in secondary storage. The amount of information written to secondary storage will depend on the particular logging scheme supported. The expense and overhead of recording these entries should, however, be less than checkpointing the entire object, since only the modifications made to an object are written. Enough information is maintained in the log so that an object that has failed can be restored to its state as of the last commit. One drawback of the log scheme is that the larger the log gets, the greater the overhead of object maintenance and recovery.

The simplest type of log is a *redo log*. A redo log records a base checkpoint for every persistent object, the changes made to each object since the base checkpoint, and the current status of the actions that made these changes. When an object is modified and the action that made the modification commits, the relative changes made to the object are recorded in the log. For example, a redo log can record the new values of the updated data or can record the operations

performed on the objects. If an object fails, it is restored to its last consistent state by reperforming the modifications made since the base checkpoint by actions that have committed. Periodically a new base checkpoint is recorded for each object, and old log entries are cleared from the log. The redo log is a straightforward and efficient scheme.

Another type of log is the *commit log* [Oki et al. 85]. A commit log records the states of all objects modified by actions that have been or are in the process of being committed. A log maintains an ordered list of data entries and outcome entries. A data entry records the checkpoint of an object. An outcome entry records the last known stage of the commit procedure performed by an action: precommit, commit, or abort.

In the precommit stage of the commit procedure, every object that was modified by the action being committed is checkpointed and written to the log, each in its own data entry. Also written is a precommit outcome entry for the action. In the commit stage, a commit outcome entry for the action is written to the log. If the action aborts, an abort outcome entry for the action is written to the log. Periodically, the commit log is cleared of all entries that are outdated.

5.2 Processors

Administrating the use of the processors is a very important function of a DOBPS. The primary goal of managing the processors is to maximize the throughput rate of the system by minimizing the time objects have to wait to receive processor service. The task of assigning objects to processors is made difficult by two partially conflicting goals: First, objects should be assigned to different, lightly loaded processors so they can execute concurrently; second, objects that interact frequently should be assigned to the same or nearby processors to reduce their communication costs. Thus, the benefit of executing the objects of a program on multiple processors is partially offset by the additional cost of intermachine communication. For optimal performance, the objects of an object-based program should be assigned to a group of closely spaced, lightly loaded processors.

5.2.1 Object Scheduling

Whenever a new object is created or an inactive object is activated it must be assigned to a processor. A new object is usually assigned to the processor on which it is created; however, the system may permit it to be created on a remote processor. An object that is activated can typically be reassigned to any processor of the same type as the one on which it was originally created. Notable exceptions are *immobile objects* such as objects whose locations are encoded within their identifiers. Such objects are always reassigned to the same processors on which they were originally created.

The object scheduling scheme of a DOBPS may be either *explicit* or *implicit*. In the explicit scheme, the user is responsible for specifying the processor to which an object is to be assigned. In the implicit scheme, the system is responsible for determining where to assign the objects. There are a few practical implicit object scheduling algorithms, including wave scheduling [Van Tilborg and Wittie 1981; Wittie and Van Tilborg 1980], contract bidding [Smith 1979], and tokens [Tripathi and Huang 1986]. The more complex schemes examine the loads of the processors to determine the best location to place an object —typically the processor with the lightest load. When a group of interacting objects is created, such as those in an object-based program, a cluster of lightly loaded processors is found and each object is assigned to one of the processors. One drawback of using load information to determine where to assign objects is that obtaining and maintaining accurate load information can be relatively expensive. If accurate information is not maintained, there is the possibility that a processor, which is observed to be lightly loaded, may be assigned a number of objects and subsequently become overloaded.

5.2.2 Object Mobility

An *object migration* scheme[6] permits objects to move or migrate from one processor to another at any time, in some cases even while they are in the middle of servicing an invocation. The work performed by an object that is moved is not lost, nor is any action that accessed the object aborted. Two advantages of object migration are increased performance and improved availability. For example, objects may be moved from a heavily loaded processor to one with a lighter load or from a processor scheduled to be shut down for maintenance to another available one. It also enables objects that interact heavily to be moved to the same workstation so future communication costs can be reduced.

There are a number of process migration algorithms, including adaptive bidding [Stankovic and Sidhu 1984] and pairing [Bryant and Finkel 1981]. Migrating an executing process from one processor to another in a conventional distributed system is an extremely difficult task. Although it is not difficult to move the code being executed, the main problem is moving the machine-dependent information such as the values of running clocks, the logical communication paths, and the data structures maintained in memory. A DOBPS simplifies many of these problems because objects clearly define the entities that can be moved and encapsulate the components that must be moved as a unit. Furthermore, machine-dependent information is usually kept to a minimum, and the property of location transparency permits a system to determine the new location of an object that has moved automatically.

Nevertheless, there are a number of problems with object migration. First, the cost of moving an object may outweigh the benefits of the move. Moving an object from one workstation to another can be an expensive task, and an object cannot do any processing while it is being moved. Second, invocation requests sent to an object while it is being moved must be accepted by the system and forwarded to the object when it becomes active again. Third, an object should not be continuously moved about the network, otherwise it will get little processing done. Finally, replicas of an object should not be moved to the same processor.

Object migration mechanisms typically attempt to reduce the loads on heavily loaded processors. When the load of a processor exceeds some limit, the system attempts to find a suitable, less-loaded processor on which to move some of the active objects. Nearby processors are usually searched before those further away. This minimizes the distance an object is moved so it will remain relatively close to the objects with which it interacts. If an appropriate processor is found, the system determines which objects are to be moved. This decision may be based on the size of an object, its estimated remaining processing time, the number of times it has already been moved, or the overhead of moving the object. These objects are suspended, moved, then resumed.

5.3 Overview of Resource Management in Existing Systems

5.3.1 Amoeba

Amoeba represents each object in memory using the multiple versions approach and in secondary storage using the history of checkpoints scheme because it supports an optimistic concurrency control scheme. Special paging hardware is supplied to increase the performance of the system. This enables the system to support a demand paging scheme so only those pages that are required are loaded into memory. Furthermore, only those pages that are modified need to be written back to secondary storage. When an action examines or modifies an object, a page-for-page copy of its most recent version is created in memory. When an

[6]Sometimes referred to as *process migration* in non-object-based systems.

action attempts to commit, the old checkpoints of the object are examined to determine if there is a serialization conflict (see Section 3.5). If there is no conflict, the modifications are made permanent by writing the modified version of the object to secondary storage and making it the new current version.

Amoeba maintains a collection of shared processors called a processor pool from which a client may dynamically request a number of idle processors; however, it cannot specify particular processors. When a processor is no longer required, it is returned to the pool.

5.3.2 Argus

Argus represents each object in memory as a stack of versions and in secondary storage using the commit log scheme. To reduce the size of the commit log, periodically a new log is created by taking a workstation-wide checkpoint of all active objects. An object scheduling scheme is supported. Objects are immobile since their locations are encoded within their identifiers.

5.3.3 CHORUS

CHORUS represents each object in memory using the single version approach and in secondary storage using the pure checkpoint scheme. An object scheduling scheme is supported.

5.3.4 Clouds

Clouds represents each object in memory using the single version approach and in secondary storage using the pure checkpoint scheme. A paging scheme is supported. An object is maintained in secondary storage as a core image that is paged into memory on demand. The paging hardware permits a process to have two segments: a data segment used to store the stack of the process and a code segment used to store the object. When a process makes an invocation on an object, the code segment of the process is switched so the new object is mapped into the address space of the process. An object scheduling scheme is supported.

5.3.5 Eden

Eden represents each object in memory using the single version approach. In secondary storage, objects are represented using the history of checkpoints scheme and maintained by a file server. Each object is managed by a version manager that controls access to the most recently committed version and all uncommitted versions of the object. Objects are accessed via their version manager and may be opened, closed, or committed. When an action modifies an object, the object is opened and a copy of its current version is loaded into memory. When an action completes, the object is closed and a new, uncommitted version of the object is created. If another action modifies the object before it is committed (as in the case of a nested transaction), the object is reopened and a copy of the uncommitted version of the object is made. When all the actions affecting an object have completed and the action commits, the most recent uncommitted version of the object is made the current version. An object scheduling scheme is supported.

5.3.6 Emerald

Emerald represents each object in memory using the single version approach and in secondary storage using the pure checkpoint scheme.

An object migration scheme is supported. To simplify the task of moving objects, Emerald generates relocatable code and creates templates that describe the internal structure of the objects.

An object is moved according to the following procedure:

(1) All processes executing in the object are suspended.
(2) A template of the object is made.
(3) The object's state information and template are sent to the new workstation.

(4) The operating system at the new workstation rebuilds the object by allocating space for the object and copying the state into that space. Using the template, the state information is traversed and all pointers are replaced with their new addresses.
(5) Finally, the processes are resumed.

Details of this object migration scheme is given in Jul et al. [1988].

5.3.7 TABS / Camelot

TABS and Camelot represent each object in memory using the single version approach. A paging scheme is used by both. These systems differ from most DOBPSs in that an object can record its modifications to secondary storage even before the corresponding action is committed. This enables the system to record a checkpoint of an entire workstation to ensure that a globally consistent state is recorded.

An object is represented in secondary storage using the undo/redo log scheme, a variation of the redo log scheme. An undo/redo log maintains enough information so the effects of an aborted action can be undone and the effects of a committed action can be redone. TABS provides two types of logging schemes: one that uses old value/new value entries and one that uses operation entries. An old value/new value entry records both the old and new values of a modified page. An operation entry records the name of the operation invoked and enough information to invoke it again.

Camelot also provides two types of logging schemes: one that uses old value/new value entries and one that uses new value entires. A new value entry records the new value of a modified page. Each action may specify the logging scheme that is to be used for the action. To perform a checkpoint, the system informs all of the objects that reside in its workstation to suspend themselves in their next consistent state. An object that reaches a consistent state suspends its activities and notifies the system. When all objects have suspended themselves, a list of the pages that reside in memory and the status of all executing actions are recorded in the log. The activities of the system are then resumed.

See Figure A5 for a summary of the Resource Management schemes supported by each system.

6. CONCLUSION

Designing a distributed, object-based programming system is a complex and difficult task. Careful thought and planning must go into determining what functionality and features to provide. Unfortunately, at this time there is no single set of features that can be provided to solve the needs of all DOBPSs. Many of the features provided by a particular DOBPS will depend on the intended application of the system. Fortunately, the development of a DOBPS is simplified by the use of objects, since objects are autonomous entities that serve well as the units for protection, recovery, security, synchronization, and mobility.

The prime advantage to using a DOBPS is that it alleviates many of the problems associated with creating and executing distributed programs. The object abstraction serves as a bridge between a programmer and a machine by creating a common primitive that reduces the complexity of the human being/machine interface. This is the fundamental characteristic of these systems. A DOBPS simplifies the programming language interface to permit a programmer to express his or her ideas in a program more conveniently. It also simplifies the operating system interface to enable a program to execute efficiently on a machine. This is a trend away from what has traditionally been done: Instead of users complying to the demands of machines, machines are now being built to comply with the demands of their users.

APPENDIX

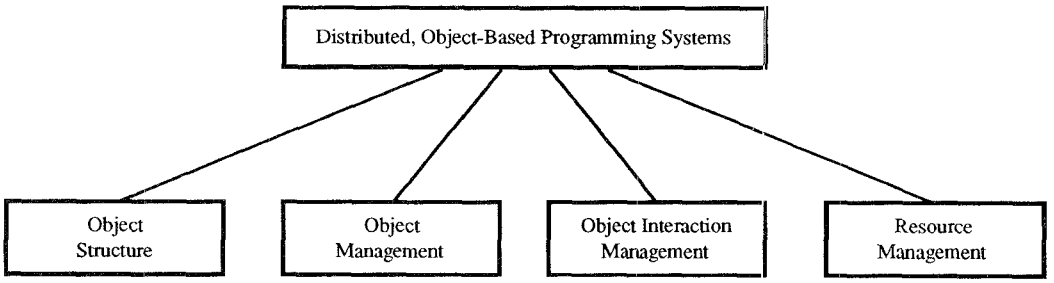
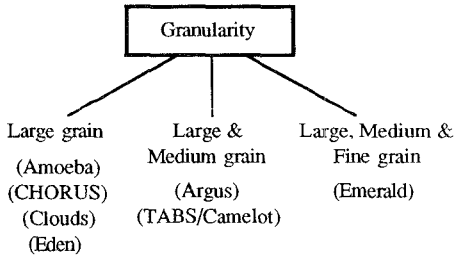


Figure A1. Classification categories.



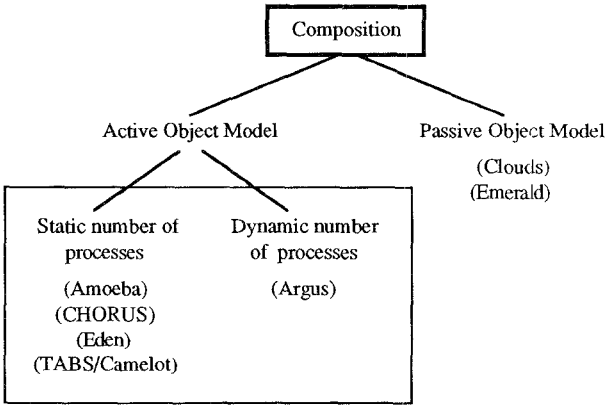


Figure A2. Object structure.

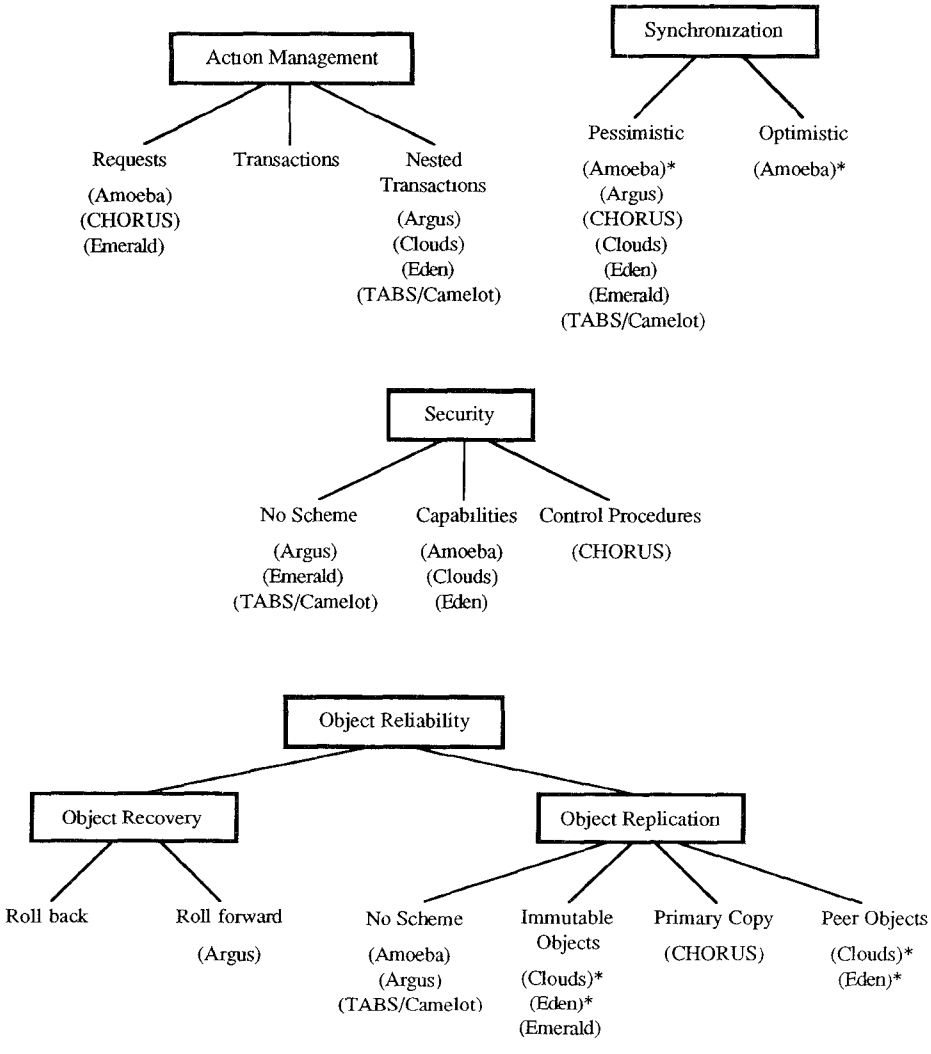


Figure A3. Object management.

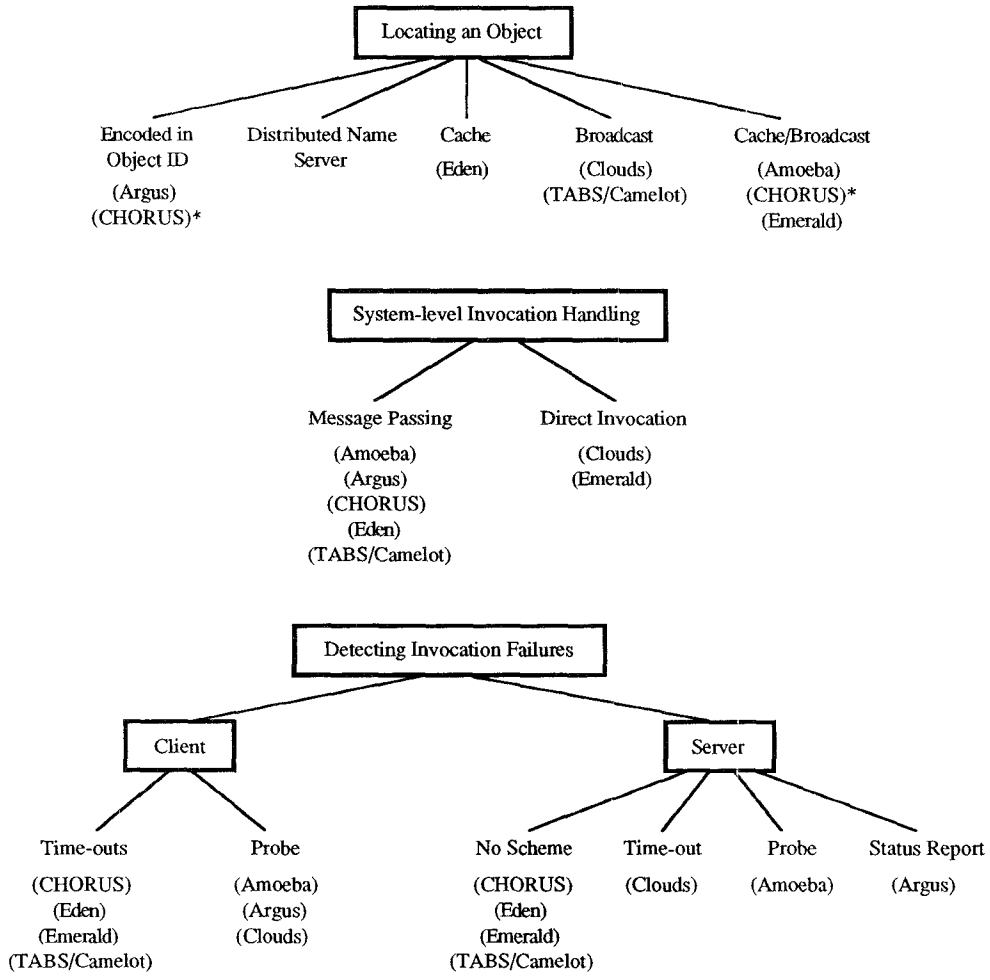


Figure A4. Object interaction management.

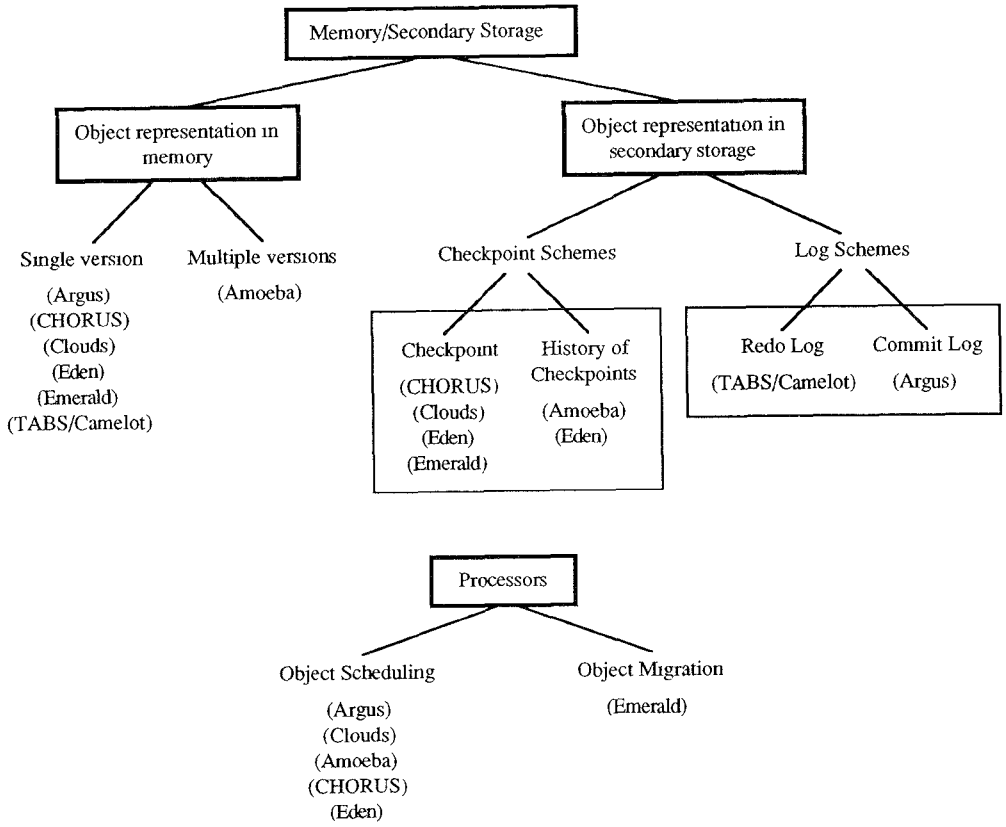


Figure A5. Resource management.

REFERENCES

AHAMAD, M., AND DASGUPTA, P. 1987. Parallel execution threads: An approach to fault-tolerant actions. Tech. Rep GIT-ICS-87/1 School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Ga.

AHAMAD, M., DASGUPTA, P , LE BLANC, R J., AND WILKES, C T. 1987. Fault tolerant computing in object based distributed operating systems. In *IEEE 6th Symposium on Reliability in Distributed Software and Database Systems.* (Mar.), pp 115-125.

ALSBERG, P A., AND DAY, J. D. 1976. A principle for resilient sharing of distributed resources. In *Proceedings of the IEEE 2nd International Conference on Software Engineering.* pp. 562-570.

ALMES, G. T., BLACK, A. P., LAZOWSKA, E. D., AND NOE, J. D. 1985. The Eden system: A technical review *IEEE Trans. Softw. Eng. SE-11,* 1 (Jan.), 43-58.

ANDREWS, G. R., OLSSON, R. A , COFFIN, M., ELSHOFF, I., NILSEN, K., PURDIN, T., AND TOWNSEND, G. 1988. An overview of the SR language and implementation. *ACM Trans. Program. Lang. Syst. 10,* 1 (Jan.), 51-86.

BANINO, J. S., AND FABRE, J C. 1982. Distributed coupled actors: A CHORUS proposal for reliability. In *IEEE 3rd International Conference on Distributed Computing Systems.* (Oct.), pp. 128-134.

BANINO, J. S., FABRE, J. C., GUILLEMONT, M., MORISSET, G., AND ROZIER, M. 1985. Some fault-tolerant aspects of the CHORUS distributed system. In *IEEE 5th International Conference on Distributed Computing Systems* (May), pp. 430-437.

BERNSTEIN, P. A., AND GOODMAN, N. 1981. Concurrency control in distributed database systems. *ACM Comput. Surv. 13,* 2 (Jun.), 185-221.

BERNSTEIN, P. A., AND GOODMAN, N. 1984. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. Database Syst. 9,* (Dec.), 596-615.

BLACK, A., HUTCHINSON, N., JUL, E., AND LEVY, H. 1986A. Object structure in the Emerald system. Tech. Rep. 86-04-03. Department of Com-

puter Science, University of Washington, Seattle, Wash.

BLACK, A., HUTCHINSON, N., JUL, E., LEVY, H., AND CARTER, L. 1986B. Distribution and abstract types in Emerald. Tech. Rep. 86-02-04. Department of Computer Science, University of Washington, Seattle, Wash.

BRYANT, R. M., AND FINKEL, R. A. 1981. A stable distributed scheduling algorithm. In *IEEE Proceedings of the 2nd International Conference on Distributed Computing Systems*. pp. 314-323.

COHEN, E., AND JEFFERSON, D. 1975 Protection in the Hydra operating system. In *ACM Proceedings of the 5th Symposium on Operating System Principles 9*, 5 (Nov.), 141-160.

DASGUPTA, P. 1986. A probe-based monitoring scheme for an object-oriented, distributed operating system. In *ACM Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*. pp. 57-66.

DASGUPTA, P., LEBLANC, R., AND APPELBE, W. 1989. The Clouds distributed operating system. Functional description, implementation details and related work In *IEEE 8th International Conference on Distributed Computing Systems*. San Jose.

DOD 1980. *Ada Reference Manual*, U. S. Department of Defense.

EPPINGER, J. L., AND SPECTOR, A. Z. 1985. Virtual memory management for recoverable objects in the TABS prototype. Tech. Rep. CMU-CS-85-163, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Penn.

GIFFORD, D. K. 1979. Weighted voting for replicated data. In *ACM Proceedings of the 7th Symposium on Operating System Principles* (Dec.), pp. 150-162.

GOLDBERG, A., AND ROBSON, D. 1983. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Mass.

GRAY, J. N. 1978. Notes on database operating systems. In *Lecture Notes in Computer Science*, 1978, Springer, New York, 1978, pp. 393-481.

GRAY, J. N. 1980. A transaction model. Tech. Rep. RJ2895, IBM Research Laboratory, San Jose, Calif.

GUILLEMONT, M., AND MARTINS, J. L. 1987. CHORUS: A new UNIX for the distribution age. Submitted for publication. Currently available from the authors at INRIA.

JONES, A. K. 1976. The narrowing gap between language systems and operating systems. Computer Science Research Review 1975-1976 Carnegie-Mellon University, p. 17.

JUL, E., LEVY, H., HUTCHINSON, N., AND BLACK, A. 1988. Fine-grained mobility in the Emerald system. *ACM Trans. Comput. Syst. 6*, 1 (Feb.), 109-133.

KORTH, H. F. 1983. Locking primitives in a database system. *J. ACM 30* 1 (Jan.), 55-79.

LIEBERHERR, K. J., AND HOLLAND, I. M. 1989. Assuming good style for object-oriented programs. *IEEE Softw.* (Sept.), 38-48.

LISKOV, B. 1988. Distributed programming in Argus. *Commun. ACM 31*, 3 (Mar.), 300-312.

LISKOV, B., CURTIS, D , JOHNSON, P., AND SCHEIFLER, R. 1987. Implementation of Argus. In *ACM Proceedings 12th Symposium on Operating System Principles* pp. 111-122.

MCKENDRY, M. S., AND HERLIHY, M. 1985. Time driven orphan elimination. Tech. Rep. CMU-CS-85-138. Computer Science Department, Carnegie-Mellon University, Pittsburgh, Penn.

MOSS, J. E. 1985. Nested transactions: An approach to reliable distributed computing. Tech. Rep. MIT/LCS/TR-260, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass.

MULLENDER, S. J., AND TANENBAUM, A. S. 1985 A distributed file service based on optimistic concurrency control. In *ACM 10th Symposium on Software Principles*.

MULLENDER, S. J., AND TANENBAUM, A. S. 1986 The design of a capability-based distributed operating system. *Comput. J. 29*, 4 (Aug.).

NICOL, J. R., BLAIR, G. S., AND WALPOLE, J. 1987. Operating system design: Towards a holistic approach? *ACM Operat. Syst. Rev. 21*, 1 (Jan.), 11-19.

OKI, B. M., LISKOV, B. H., AND SCHEIFLER, R. W. 1985. Reliable object storage to support atomic actions. In *ACM Proceedings of the 10th Symposium on Operating System Principles*. pp. 147-159.

PITTS, D. V., AND DASGUPTA, P. 1988. Object memory and storage management in the Clouds kernel. In *IEEE 8th International Conference on Distributed Computing Systems* (San Jose).

POWELL, M. L., AND PRESOTTO, D. L. 1983. Publishing: A reliable broadcast communication mechanism. *ACM Operat. Syst. Rev. 17*, 5, 100-109

PU, C., NOE, J. D., PROUDFOOT, A. 1986 Regeneration of replicated objects: A technique and Its Eden implementation. In *IEEE Proceedings 2nd International Conference on Data Engineering*. (Feb.), pp. 175-187.

ROZIER, M., AND MARTINS, J. L. 1987. The CHORUS distributed operating system: Some design issues. In *Distributed Operating Systems. Theory and Practice*. Springer-Verlag, Berlin, Heidelberg, pp. 262-287,

SMITH, R. G. 1979 The contract net protocol: High-level communication and control in a distributed problem solver. In *IEEE Proceedings of the 1st International Conference on Distributed Computing Systems*. pp. 185-192.

SPAFFORD, E. H. 1987. Object operation invocation in Clouds. Tech. Rep. GIT-ICS-87/14. School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Ga.

SPECTOR, A. Z. 1987. Distributed transaction processing and the Camelot system. Tech. Rep. CMU-CS-87-100. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Penn.

SPECTOR, A. Z., DANIELS, D. S., DUCHAMP, D., EPPINGER, J. L., AND PAUSCH, R. 1985. Distributed transactions for reliable systems. Tech. Rep. CMU-CS-85-117, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Penn.

SPECTOR, A. Z., THOMPSON D. S., PAUSCH, R. F., EPPINGER, J. L., DUCHAMP, D., DRAVES, R. P., DANIELS, D. S., AND BLOCH, J. J. 1987. Camelot: A distributed transaction facility for mach and the internet: An interim report. Tech. Rep. CMU-CS-87-129. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Penn.

SPECTOR, A. Z., BLOCH, J. J., DANIELS, D. S., DRAVES, R. P., DUCHAMP, D., EPPINGER, J. L., MENEES, S. G., AND THOMPSON, D. S 1986. The Camelot project. Tech. Rep. CMU-CS-86-166. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Penn.

STANKOVIC, J. A., AND SIDHU, I. S. 1984. An adaptive bidding algorithm for processes, clusters and distributed groups. In *IEEE Proceedings of the 4th International Conference on Distributed Computing Systems*. pp. 49–59.

STROUSTRUP, B. 1986. *The C + + Programming Language*. Addison-Wesley, Massachusetts.

SVOBODOVA, L. 1984. File servers for network-based distributed systems. *Comput. Surv. 16* 4 (Dec.), 353–398.

TANENBAUM, A. S., AND VAN RENESSE, R. 1987. Reliability issues in distributed operating systems. In *IEEE 6th Symposium on Reliability in Distributed Software and Data Base Systems* (Mar.).

TANENBAUM, A. S., MULLENDER, S J., AND VAN RENESSE, R. 1986. Using sparse capabilities in a distributed operating system. In *IEEE Proceedings of the 6th International Conference on Distributed Computing Systems*. (May), pp. 558–563.

TANENBAUM, A. S., AND VAN RENESSE, R. 1985. Distributed operating systems. *ACM Comput. Surv. 17* 4 (Dec.), 419–470.

TRIPATHI, S. K., AND HUANG, S. 1986 Distributed resource scheduling for a large scale network of processors: HCSN. In *IEEE Proceedings of the 6th International Conference on Distributed Computing Systems* (May), pp. 321–327.

VAN TILBORG, A. M., AND WITTIE, L. D., 1981. Wave scheduling: Distributed allocation of task forces in network computers. In *IEEE Proceedings of the 2nd International Conference on Distributed Computing Systems*. pp. 337–347.

WALKER, E. F. 1984. Orphan detection in the Argus system. Tech. Rep. MIT/LCS/TR-326. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass.

WEGNER, P. 1987. Dimensions of object-based language design. In *ACM Proceedings of the Conference on Object Oriented Programming Systems. Languages and Applications* (Oct.), pp. 168–182.

WIRTH, N. 1985. *Programming in Modula-2*. Springer-Verlag, New York, 3rd ed.

WITTIE, L. D., AND VAN TILBORG, A. M. 1980. MICROS: A distributed operating system for MICRONET, a reconfigurable network computer. *IEEE Trans. Comput. C-29* (Dec.), 1133–1144.

Received September 1989, final revision accepted July 1990