Issues in Designing a Distributed, Object-Based Programming System

by

Roger Steven Chin

B.Sc., The University of British Columbia, 1986

A Thesis Submitted in Partial Fulfillment

of the Requirements for the Degree of

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

Department of Computer Science

We accept this thesis as conforming

to the required standard

The University of British Columbia

October 1988

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date Oct. 14, 1988

Abstract

Objects are entities which encapsulate data and those operations which manipulate the data. A distributed, object-based programming system (or DOBPS) is a distributed operating system which has been designed to support an object-based programming language and, in particular, an object abstraction. DOBPSs have the benefits of simplifying program construction and improving the performance of programs by providing efficient, system-level support for the abstractions used by the language. Many DOBPSs also permit hardware and software failures to be tolerated.

This thesis introduces a definition for the term "distributed, object-based programming system" and identifies the features, that are related to objects, which are required by an operating system of a DOBPS. A classification scheme is presented that categorizes and characterizes these features to permit a number of implementation techniques to be easily examined, compared, and contrasted.

Table of Contents

List of Tables

List of Figures

Acknowledgement

There are so many people to thank and so little room to thank them in.

Most of all, I am extremely grateful to my supervisor, Dr. Samuel Chanson, for his patience, understanding, and encouragement.

I would also like to thank my parents and all my friends for their support, allowing my years at U.B.C. to be enjoyable and memorable ones.

Chapter 1. Introduction

Traditionally, a program has been a single entity that is executed sequentially on a single processor or on a set of tightly-coupled processors such as a multiprocessor. However, many researchers now believe that a program should be composed of a set of interacting modules, or objects, that execute concurrently on a collection of loosely-coupled processors. Their rationale for this opinion is three-fold: first, a program's development should be simplified when it has a modular structure; second, a program's performance should improve when it is executed concurrently on multiple processors; third, a program should not fail as a result of the failure of a processor which it is not using. To assist the creation and execution of these types of programs, *object-based programming languages* and *distributed operating systems* have been developed. An object-based programming language encourages a user to design and create a program as a collection of autonomous components, while a distributed operating system enables a collection of workstations or personal computers to be treated as a single entity. The amalgamation of these two concepts has

resulted in systems which shall be referred to in this thesis as *distributed, object-based programming systems* (or *DOBPSs*).

Designing and developing a DOBPS is an extremely difficult task due to the novelty and complexity of the subject. An especially important and difficult issue is designing the distributed operating system. The primary problem is determining the features and functions which should be provided by the operating system to best support the object-based programming language. The purpose of this thesis is to simplify this problem by providing a scheme which indicates and classifies the numerous features that can be found in these types of operating systems. This enables the features to be studied individually and permits a number of implementation schemes to be easily examined, compared, and contrasted.

1.1. History of the Object Abstraction

In the last two decades, the concept of object abstraction has appeared in many areas of computer science, including: programming languages, databases, graphics, operating systems, and hardware. The discussion of objects in this thesis, however, will be limited to the topics of object-based programming languages, operating systems, and hardware.

The concept of object abstraction can be traced to the Simula programming language [Dahl 66, Birtwistle 73], and the Hydra multiprocessor operating system [Wulf 74, Cohen 75, Jones 75]. Simula is usually credited with first introducing the notion of providing a language feature which enabled the encapsulation of data and the procedures which acted upon the data. The only way data could be examined or manipulated was by calling one of its procedures. Initially, objects were viewed strictly as a protection mechanism, a way to encourage the use of structured programming and abstract data types. Hydra later enhanced this idea by shifting the management of objects from the language to the operating system. This gave the operating system total control over the

objects, enabling it to provide a more uniform environment and increased data security. Every resource of the system whether physical, such as an I/O device, or logical, such as a file, was treated as an object. Furthermore, access to the objects was rigorously controlled by the system. Simula and Hydra can probably be credited with creating the foundation from which all other object-based languages and programming systems were derived.

Object-based programming languages initially had the problem that the traditional operating systems they executed on rarely provided the appropriate features and functions they required. This resulted in a substantial reduction in the performance of these languages, due to the classical problem of the operating system's generality leading to its overall inefficiency. Therefore, a user of one of these languages was faced with the trade-off of a gain in functionality at the expense of performance.

To improve the performance of these languages, special purpose operating systems were developed so that system-level support for objects could be provided. This enabled the object management tasks to be handled more efficiently and effectively. An example of this type of programming system, and the one to which all others are usually compared, is Smalltalk [Goldberg 83]. Part of the popularity of Smalltalk is due to the fact that it supports one of the most complete and consistent uses of object abstraction. That is, absolutely all data entities, whether system or user defined, are treated as objects. A completely uniform computing environment is provided because everything acts and interacts in the same manner.

To improve the performance of these programming systems even further, special purpose hardware was developed. Some examples of this type of hardware include IBM's System/38 [Soltis 79] and Intel's iAPX-432 [Kahn 81, Zeigler 81]. This special hardware enabled efficient machine-level support for objects to be provided. Their use, however, has been limited to all but a few systems.

More recently, decentralized versions of these programming systems have been developed which enable a user to take advantage of a multiple machine environment. These systems have the potential for much greater performance than their centralized counterparts since they permit concurrent processing to take place. Examples of these systems include: Amoeba [Tanenbaum 81, Mullender 85, Mullender 86, Tanenbaum 86, Tanenbaum 87] at Vrije Universiteit; Argus [Liskov 83a, Liskov 83b, Walker 84, Oki 85, Liskov 87, Liskov 88] at M.I.T.; CHORUS [Banino 82, Banino 85, Guillemont 87, Rozier 87] at I.N.R.I.A.; Clouds [Spafford 84, Dasgupta 85, McKendry 85, Dasgupta 86, Ahamad 87a, Ahamad 87b, Spafford 87, Dasgupta 88, Pitts 88] at Georgia Tech.; Eden [Lazowska 81, Jessop 82, Almes 85, Black 85, Pu 86] and its successor Emerald [Black 86a, Black 86b, Jul 88] at the University of Washington; and TABS [Schwarz 84, Spector 84, Eppinger 85, Spector 85] and its successor Camelot [Spector 86, Spector 87a, Spector 87b] at C.M.U.. These systems will be discussed in detail in subsequent sections of this thesis.

1.2. Motivation

This thesis was motivated by two problems. The first problem is that the extensive use of object abstraction in the different areas of computer science has led to the overuse and misuse of the term "object". This is especially true in the area of operating systems where a wide spectrum of "object" models exist. For example, it is the opinion of some people that the processes supported by the V System [Cheriton 88] and the files supported by the LOCUS system [Walker 83] are objects, while others believe that only systems such as Smalltalk support objects. In an attempt to alleviate this problem, Wegner [Wegner 87] defines a set of terms to describe object-based languages. Unfortunately, there is no similar set of terms which can be used to describe DOBPSs.

The second problem to motivate this thesis, as was stated previously, is the difficult task of designing an operating system for a DOBPS. Currently there is no established methodologies or

guidelines which can be followed while developing a system of this type. A system designer can only draw upon his own knowledge and perhaps seek the advice of others who have created similar systems. Fortunately, information regarding the design of existing systems is widely described in the literature. Unfortunately, the amount of information available can easily become overwhelming. Adding to this difficulty are two complications: first, each operating system usually provides its own set of features and functions in order to satisfy its own set of goals; second, the terminology used to describe a particular concept or feature is not always consistent between systems. As a result, the task of determining the features an operating system should provide is often difficult.

The purpose of this thesis is to solve both of these problems. Its intention is to explicitly define the term "distributed, object-based programming system" while presenting a standard set of terms which can be used to describe the features supported by a DOBPS. This thesis will also categorize and characterize the important features required by an operating system of a DOBPS. Finally, it can serve as a reference to the methods by which these features are implemented by a number of existing systems.

1.3. Thesis Overview

The remainder of this thesis is organized as follows.

Chapter 2 describes the fundamental features and concepts which characterize a DOBPS. Some of the terminology and definitions are introduced, and a brief overview of our classification scheme is given.

Chapters 3 to 6 present a detailed description of our classification scheme. The various features found in an operating system of a DOBPS are categorized and examined. To conclude the

description of each feature, a brief statement preceded by either a "√" or a "x" may be given describing the other features which should or should not be provided by a DOBPS which supports this feature. In addition, each of these chapters will conclude with a brief description of the manner in which a number of existing systems provide the corresponding features. The omission of some details and descriptions is due to the fact that the published reports available to us at this time do not provide the information.

Chapter 3 defines the types of objects and the object models which can be supported by a DOBPS. Chapter 4 describes the features which can be supplied to manage the objects of a system, including: synchronization schemes, security schemes, and object reliability schemes. Chapter 5 describes the features to manage object invocations, including: system-level invocation schemes, object location schemes, and invocation failure detection schemes. Chapter 6 concludes the classification scheme by describing the features to manage the physical resources of a system, including: the primary memory, secondary storage devices, the processors, and the workstations.

Chapter 7 presents an example of the design of a new DOBPS. It briefly describes the decisions we made for selecting the features it supports.

Finally, chapter 8 concludes the thesis. It presents some topics which require further research and some of the issues involved.

The emphasis of this thesis is placed upon a description of DOBPS issues which are directly related to object abstraction. Those issues which are indirectly related to object abstraction will be discussed briefly; however, appropriate references will be cited. Issues dealing with operating system development in general, such as kernel design, will not be discussed at all. A description of the object-based programming language design aspect of a DOBPS will be also limited.

Furthermore, it is our opinion that the concept of inheritance (see §2.2.) is a feature which should be supported by a programming language rather than an operating system. We believe that implementing inheritance at the operating system level is a difficult task. This claim appears to be substantiated by the observation that very few existing operating systems attempt to support the object-oriented paradigm and implement inheritance. Consequently, the scope of this thesis will be to describe object-based systems rather than object-oriented systems.

Chapter 2.
The Fundamental Concepts

This chapter describes the fundamental concepts and features which in our opinion form the basis of almost all distributed, object-based programming systems. We first define our notion of an object and then build upon it to define the concepts of an object-based programming language and a DOBPS. The chapter concludes with a comparison of DOBPSs to conventional systems to show their relative advantages and disadvantages.

2.1. Objects

An *object* is the direct extension of the notion of an abstract data type. It is an encapsulation of some private *state information*, or data, and a set of associated *operations*, or procedures, that manipulate the data. (See Figure 2.1). An object groups together a data structure and the procedures that act on it so that collectively they can be treated as a single entity. An object's state is completely protected and hidden from all other objects. The only way that it can be examined or modified is via a request to one of the object's operations, referred to as an *operation invocation*.

This provides a well defined interface for each object. It enables the specification of an object's operations to be made public, while at the same time it keeps the implementation of the operations and the representation of its state information private.

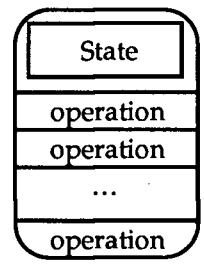

Figure 2.1: An object

Typically, an operation is not accessed directly, as in the case of a conventional procedure call. Instead, when a *client* makes an operation invocation, the corresponding *server* object usually performs the operation on the client's behalf. The interactions between a client and a server object are as follows.

(a) The client presents an invocation request together with a list of parameters to the appropriate object specifying the operation to be invoked.

(b) The server object accepts the request, locates and performs the specified operation.

(c) While the server object is performing the operation, it may make other operation invocations, possibly to other objects. These objects may in turn make invocations on others, and so on. A chain of related invocations is called an *action*. (See Figure 2.2).

(d) Finally, when the operation completes, the server object returns a result back to the client.

Invocations appear to be nothing more than a dynamic (albeit expensive) procedure calls; however, they serve the very important purpose of hiding the internal workings of the objects from their clients. For example, a client never knows how an invocation request is processed by a server, the

client has to assume that the specified operation was performed properly and that the result is correct.

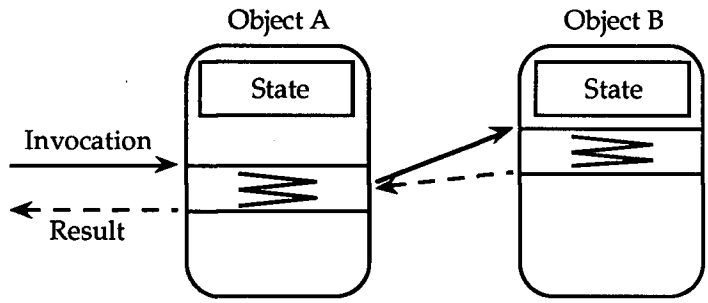

Figure 2.2: An action

2.2. Object-Based Programming Languages

A programming language is defined as being *object-based* if it supports object abstraction as a language feature and *object-oriented* if it also supports the concept of *inheritance* or *subclassing*. Inheritance is a very powerful feature which enables new objects to be developed from existing objects simply by specifying how they differ. A group of objects that have the same set of operations and the same state representations are considered to be of the same *class*. Each object of a class is referred to as an *instance* of that class. Relationships between the classes can be built by organizing them into hierarchical structures. A class may *inherit* the attributes such as the state information and the operations of another class, a *superclass*; it may also have its attributes inherited by another, a *subclass*. For example, if class A inherits the attributes of class B, then B is a superclass of A and A is a subclass of B. Each class typically may have only one superclass. Inheritance permits a subclass to provide all the operations of its superclass in addition to the ones it provides to specialize its behavior. Furthermore, a subclass may be permitted to directly

examine and manipulate the state information of a superclass. The main advantage of inheritance is that it can greatly reduce the amount of code which has to be written.

According to these definitions, C++ [Stroustrup 86], SR [Andrews 88] and Smalltalk are considered as object-oriented languages. Ada [DOD 80] and Modula-2 [Wirth 85] on the other hand, are considered as object-based languages.[1]

Object-based programming languages encourage their users to design and develop a program consisting of multiple, interacting, autonomous objects. (See Figure 2.3). However, this philosophy of dividing a program into multiple components is not new. In fact, many of these ideas come from traditional software engineering principles which 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. The design decisions and data structures should be hidden.
- *Structured.* A large program should be decomposed into components that are of a manageable size. Furthermore, the components should have well defined relationships.
- *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.

The benefits of this type of design is the creation of an easily coded, tested, and understood program. This is exactly the style of programming that is encouraged by an object-based programming language.

---

[1] A detailed discussion of this topic is given in [Wegner 87].

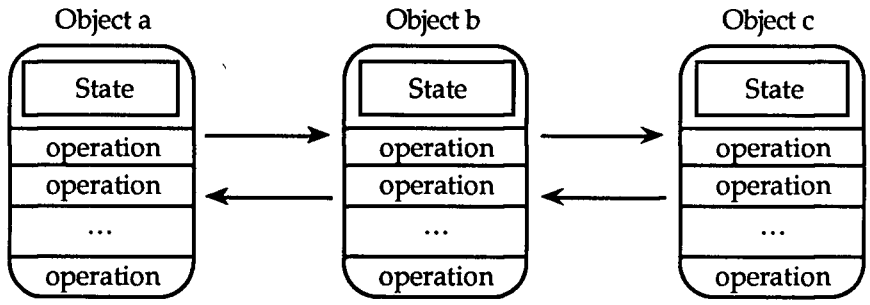

Figure 2.3: An object-based program

While this style of programming can be achieved using a traditional language such as C or Pascal, an object-based programming language permits it to be done with much less difficulty. There are two limitations of using a traditional language for this style of programming. First, subroutines are not particularly well suited for characterizing abstract entities such as objects. Second, traditional languages are generally inadequate in supporting the design of distributed programs which must deal with issues that do not arise in sequential programs, such as concurrency and reliability. Object-based programming language on the other hand, usually do not have these undesirable characteristics.

2.3. Object-Based Programming Systems

An *object-based programming system* (or *OBPS*)can best be defined as a computing environment which supplies both an object-based programming language and an operating system which supports object abstraction at the system-level, for efficiency. An advantage to using an OBPS is that it may allow objects to be shared by multiple users. In contrast, an object-base programming language does not allow objects to be shared. In addition, an OBPS can create a uniform environment by presenting every entity in the system as an object, possibly even the kernel would appear to be an object.

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

- object management,
- object invocation management, and
- resource management.

These topics are discussed briefly below and in detail in subsequent chapters of this thesis.

Typically, the kernel of an OBPS provides minimal services to enables it to be small, reliable, and efficient. For example, a kernel may provide only process, object, and communication management. The other functions of the operating system are usually provided by server objects which may reside either in kernel space or user space. The advantage of this approach is that it provides the system with greater flexibility. For example, the same operating system service may be supplied by multiple server objects, thereby allowing a program to utilize the server object best suited for it.

An operating system of an OBPS is responsible for managing both the passive entities, the objects, and the active entities, the processes, of the system. Typically, a number of processes execute within an object, each servicing an invocation made on the object. An operating system should synchronize the execution of multiple processes which attempt to access the state of the same object simultaneously, and it should provide a security scheme to ensure that only authorized clients are able to make invocations on an object.

An operating system is also responsible for managing the interactions between objects. It has to convert all invocation requests to their appropriate forms, locate the specified objects, deliver the requests, and return the results back to the client processes. It should also detect the failure of an invocation so that a client does not wait indefinitely for a result which may never arrive, and a server does not tie up valuable system resources.

Finally, an operating system is responsible for managing the physical resources of a system, including: the primary memory[2], secondary storage devices, and processors. The management of the memory and secondary storage devices is especially important because objects are swapped between the two resources. Objects which are lost if the machine in which they reside fails are said to be *volatile*. Volatile objects are temporary. They reside solely in memory and are relatively inexpensive to maintain and use. Objects which can survive the failure of their machine with a high probability are said to be *persistent*. In order for a persistent object to survive a machine failure, a version of the object has to be recorded in secondary storage. As a result, whenever a persistent object's version in memory is modified, its version in secondary storage also has to be updated. Consequently, persistent objects are more expensive to use than volatile objects. An operating system may automatically load a persistent object into memory when it is invoked and return it to secondary storage when it is no longer in use.

An OBPS usually performs all of these functions in a transparent fashion in order to shield its users from the complexity of the underlying system.

2.4. Distributed, Object-Based Programming Systems

A *distributed operating system* combines a network of independent, possibly heterogeneous (see §8.2.), workstations or personal computers[3] so that they can be treated as a single entity. A distributed system is characterized by the following three features.

- The resources of a system can be shared and accessed by a workstation regardless of their locations.

---

2 Hereafter referred to simply as memory.

3 Hereafter referred to simply as workstations.

- A distributed program may have components residing in different workstations and, in some cases, a program may be moved from one workstation to another.
- A system is able to continue running despite the failure or removal of some of its workstations, possibly at the expense of reduced functionality and/or performance.

Unfortunately, these features can be difficult to implement.

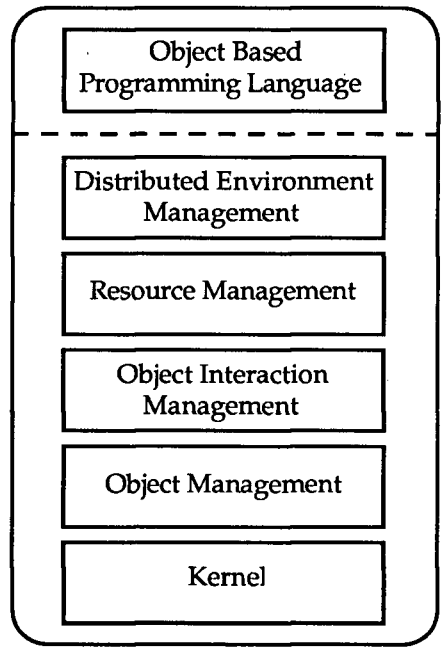

Figure 2.4: Elements of a distributed object-based programming system

A *distributed, object-based programming system* provides the same features as an object-based programming system in addition to providing a decentralized or distributed computing environment. (See Figure 2.4). Objects can be used to effectively handle some of the problems of implementing a distributed operating system. For example, objects explicitly define the

independent and autonomous entities of a system which can be distributed, recovered, and synchronized. A distributed, object-based programming system has the following characteristics[4].

- *Distribution.* A DOBPS is composed of a collection of loosely-coupled workstations connected by a network, usually a local area network. Each workstation is autonomous and executes its own copy of the operating system, thereby providing a decentralized computing environment.
- *Workstation Autonomy.* A DOBPS may permit the owner of a workstation to retain control over how the workstation and its physical resources are used.
- *Reconfiguration.* A DOBPS can dynamically adapt to changes in its hardware environment by reconfiguring itself whenever workstations are added or removed from the network. This allows modifications to be made as the demand for the system changes.
- *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. Furthermore, a network link failure can, at most, result in the loss of those workstations on the other side of the partition. In contrast, the failure of a processor in a centralized system results in the loss of the entire system.
- *Transparency.* A DOBPS may hide the complexity of the distributed environment or other underlying details from its users. For example, a DOBPS provides the feature of *location transparency* so that a user does not have to be aware of machine boundaries and the physical locations of objects in order to make an invocation on an object.
- *Object Autonomy.* A DOBPS may permit the owner of an object to specify the clients which have the authority to make invocations on the object.

---

4 Some of these definitions were derived from [MIT 86].

- *Program Concurrency.* A DOBPS can assign the objects of a program to multiple processors so that they can execute concurrently (see Figure 2.5). However, they can also execute on a single processor when others are not available.
- *Object Concurrency.* A DOBPS may permit an object to serve multiple invocation requests concurrently. This is not true concurrency, however, unless an object resides in a multiprocessor. Even if an object supports multiple processes, only one process can execute at a time on a workstation with a single processor. Typically, a process executes until it blocks, at which time another process is restarted.
- *Data Integrity.* A DOBPS ensures that an object is always in a valid state before it performs an invocation. That is, an object is always in a state which is the result of the successful termination of an operation. If for any reason an operation does not complete, the system ensures that all changes made to the object's state are undone.
- *Availability.* A DOBPS may ensure that an object is always available despite workstation failures. This is typically done by creating replicas of an object and assigning them to different workstations. Should enough workstation failures result in the object being unavailable, then the entire system is shut down and recovered in order to restore full functionality.
- *Recoverability.* A DOBPS automatically recovers from the failure of an object or the workstation on which it resides. An object is usually restored to its last consistent state and reloaded on its workstation, or possibly on a second workstation if the first has failed.
- *Improved Performance.* The response time to execute a program in a DOBPS should be lower than in a conventional system.

These topics will be discussed in greater detail in subsequent sections of this thesis.

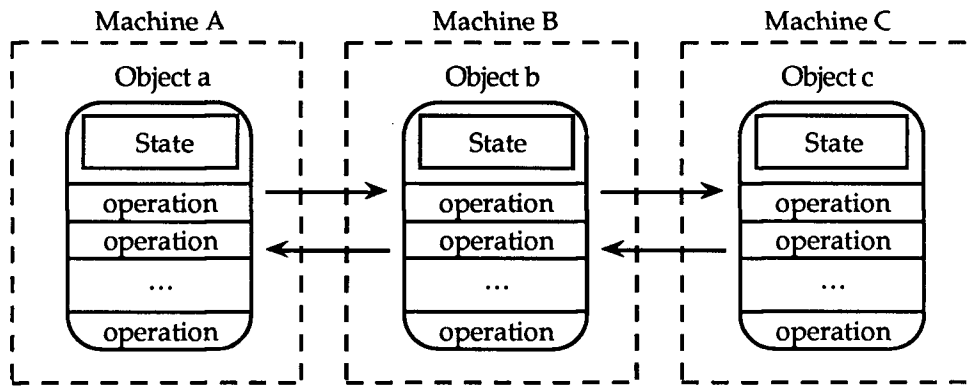

Figure 2.5: A distributed object-based program

2.5. Advantages

Distributed, object-based programming systems offer many features which make them advantageous to conventional, centralized, non object-based systems.

- Providing a programming language which eases the development of large, complex programs, especially distributed programs.
- Providing efficient, operating system level support for the abstractions supported by the programming language.
- Providing a distributed computing environment.

These features are explained in greater detail in the following paragraphs.

The primary advantage of an object-based programming language is the ease with which it allows programming to be done. Its strength is in allowing the abstractions used by a programmer to solve a problem to be easily translated into the abstractions featured by the language. An object-based programming language stresses a development methodology which emphasizes modular structure, data abstraction, and the reuse of code. Respectively, these characteristics correspond to the human approach of dividing a complex problem into less complex sub-problems, obtaining the

important information while ignoring those details which are immaterial, and drawing upon previously obtained knowledge to help solve a problem which has been at least partially solved before.

An object-based programming language encourages the decomposition of a program into multiple, autonomous components. This permits a program's framework to be uncoupled from the details of its implementation, allowing the development process to be initially considered at a higher level of abstraction. It also permits a programmer to get a conceptual overview of his program which should in turn enable him to organize its design more effectively. In addition, a modular structure localizes the impact of design decisions and permits each object to be developed independently, perhaps by different programmers. This also simplifies the tasks of modifying and maintaining a program since changes can be made in an object's implementation without affecting other objects. Finally, the development process is simplified by allowing previously defined objects to be reused easily. In summary, there are two major benefits of using object-based programming languages. First, they should result in more understandable programs which are easier to maintain; second, they should lower the cost of developing software by increasing programmer productivity. These languages simplify the development of programs by creating a close coupling between the language and the programmer.

The distinction between an operating system and the programming language it supports is also not as great as it once was. Jones observed that operating systems and programming languages were once designed and developed independently, however, a close coupling was beginning to form between them[5]. More recently, languages are now providing features that were previously provided only by the operating system, and vice versa. The primary reason for this tight coupling

---

[5] A.K. Jones, "The Narrowing Gap Between Language Systems and Operating Systems", *Computer Science Research Review 1975-1976 Carnegie-Mellon University*, p. 17.

is so that efficient low-level support can be provided for the higher-level abstractions. This is part of the process which Nicol, Blair, and Walpole call *total system design* [Nicol 87]. 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 which has the benefit of creating a more uniform and efficient system.

A major advantage of a DOBPS is its distributed computing environment. In addition to the aforementioned performance and fault tolerance advantages (see §2.4.), there is also an economic reason to explain the popularity of these systems. Recently, technology has lowered the cost of powerful workstations and personal computers, but they have not been able to keep up with the ever increasing demand for computing power. This has created a trend towards distributed computing systems which take advantage of the combined processing power of a group of workstations. Unfortunately, a major drawback of these systems is that due to their complexity, only a small number of programmers are able to take full advantage of them. Distributed, object-based programming systems alleviate this problem.

2.6. Disadvantages

Distributed, object-based programming systems are not without their disadvantages. They have a number of problems, including:

- they may be initially difficult to use,
- they may have higher overhead, and
- they require the underlying operating systems to support the object models adopted by the programming systems for efficiency.

These points are explained in detail below.

Even though object-based programming languages are designed to model human conceptions, studies have found that some people have initial difficulties in understanding and learning how to use them. O'Shea observed that the trouble was not in understanding the fundamental concepts, instead, it was in being able to apply them [O'Shea 86]. Many experienced programmers had difficulties decomposing a problem into objects and then even more trouble implementing them. This phenomenon exists because the methodologies enforced upon programmers by traditional programming languages are not applicable when using an object-based language. Furthermore, part of the problem is due to the fact that many of the concepts found in object-based languages, while not exactly novel, are unlike those found in traditional languages. O'Shea also notes that once these problems were overcome, most people liked programming with an object-based language. Fortunately, the potential benefits of using these languages far outweigh this initial cost.

The goal of a DOBPS is to transfer work from the user to the programming system. Unfortunately, a drawback of this is that relatively high levels of overhead are required to realize this goal. This overhead is a result of the high cost of managing and maintaining both the distributed environment and the objects of the system. For example, some studies [Cox 83, Cox 84] have shown that even a very efficient invocation scheme, which uses message passing, is approximately two times more expensive than a conventional procedure call. While the cost of using objects is large compared to the cost of using the data abstractions of a traditional programming language, objects are invaluable in a distributed system where data has to be shared by multiple clients and protected against hardware failures. Furthermore, this increase in overhead is offset by a substantial increase in computing power due to a more efficient operating system and the distributed computing environment. Whether this increase in computing power totally offsets the overhead of the system depends on the characteristics of the program being executed and the features provided by the DOBPS.

Another point worth noting is that DOBPSs require specially designed operating systems to run efficiently and to manage objects according to the object models adopted by the programming systems. As with anything that is built to serve a special purpose, these operating systems work well for their intended applications, however, they may not work so well in other applications. Therefore, while an operating system is well suited for the programming environment it was originally designed for, a change in the programming system may render the operating system obsolete. This lack of generality of the operating system may be a major drawback in developing DOBPSs.

2.7. Summary

A distributed, object-based programming system amalgamates an object-based programming language with a distributed operating system. This enables a DOBPS to support objects at the system-level so that they can be used efficiently.

An object-based programming language provides a means for creating and characterizing objects. Objects enable the framework of a program to be conceived and designed quickly, thus permitting a programmer to determine the feasibility of an approach early. Once this is done, the different components of the program can be designed, written, and modified independently. Therefore, an object-based language simplifies the task of translating a problem into a program.

A distributed operating system combines a group of workstations so that collectively they can be treated as a single entity. This enables the modules of a distributed program to execute concurrently on separate workstations. It also enables the system to dynamically adapt to changes in its computing environment such as workstation additions, removals, and failures.

To summarize, the fundamental concepts of a DOBPS can be defined in relation to one another. (See Figure 2.6). Furthermore, object abstraction is supported by a DOBPS by providing facilities for object management, object invocation management, and resource management.

Object	=	data + operations
Object-based Prog. Language	=	Object + programming language
Object-based Prog. System	=	Object-based Prog. Language + operating system
Dist. Object-based Prog. System	=	OBPS + multiple machine environment

Figure 2.6: Fundamental concepts

Chapter 3.
Object Structure

The structure of the objects supported by a DOBPS influences its overall design and the features which should and should not be provided. In this chapter, we define the three types of objects (namely large-grain, medium-grain, and fine-grain objects) which can be supported by these systems, and the two ways they can be composed (namely the active and the passive object models). Finally, we present a brief overview of the manner in which objects are structured in a number of existing systems.

3.1. Granularity

The relative size, overhead, and the 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 include a major component of a program, a file, or a large database. Large-grain

objects typically reside in their own address spaces. This enables the system to provide hardware protection between objects and ensures a software fault is contained within the responsible object, unless it is a catastrophic failure. Unfortunately, there are a number of drawbacks associated with providing a separate address space for each object, including: the expense of creating and managing the objects, and the added complexity of object interactions. For example, there is the added expense of context switches. Consequently, large-grain objects are very heavy-weight entities. There are two deficiencies with a DOBPS which only supports large-grain objects. The first problem is that the system's control and protection over data is at the level of the large-grain objects. This restricts the flexibility of the system and the amount of object concurrency which can be provided. The second problem is that the system does not provide a consistent data model. Larger data entities are represented as objects while 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 are data structures such as a linked list, a queue, or a small database. A single large-grain object encapsulates a number of medium-grain objects which reside in its address space. This permits a greater amount of concurrency within a large-grain object since synchronizing access to the data (see §4.2.) 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 §4.1.) can be reduced for the same reason. A drawback of using medium-grain objects, however, is the additional overhead on the system due to the greater number of objects which have to be used and managed. In addition, a consistent data model is still not provided.

To provide an even finer level of control over data and a consistent data model, 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 include data types such as a boolean or an integer provided by a conventional programming language. Each fine-grain object is encapsulated by, and resides in the address space of, a single medium-grain or large-grain object. Therefore, a large-grain object will be composed of a number of medium-grain and fine-grain objects, while a medium-grain object will be composed of a number of fine-grain objects. The major drawback of using fine-grain objects is that the overhead of managing and using objects is compounded by the large number of objects in the system. However, this approach has the benefit of providing a completely uniform environment in which every data entity, no matter how large or small, is an object.

x[6] When a DOBPS supports the large, medium, and fine-grain object model, the active object model (§3.2.1.) and the pure message passing scheme (§5.1.1.) should not be used because of the large overhead that could result from the object interactions.

√ Consequently, the passive object model (§3.2.2.) and the direct invocation scheme (§5.1.2.) should be used when the large, medium, and fine-grain object model is supported.

3.2. Composition

A main characteristic of an object's composition is the relationship between the processes and the objects of a DOBPS. The processes may either be coupled and permanently bound to the objects

6 The definition of the symbols "x" and "√" can be found in §1.3. on p. 6.

they execute in, or they may be separate and temporarily bound to the objects they invoke. These two approaches correspond to the active object model and the passive object model, respectively.

3.2.1. Active Object Model

In the *active object model*, a number of server processes are created for and assigned to each object to handle its invocation requests. Each process is bound and restricted to the particular object it is assigned to. When an object is destroyed, so are its processes. A server process is responsible for accepting an invocation request delivered to its object, executing the specified operation, and then returning a result. 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. As a result, multiple processes may be involved in performing a single action. (See Figure 3.1).

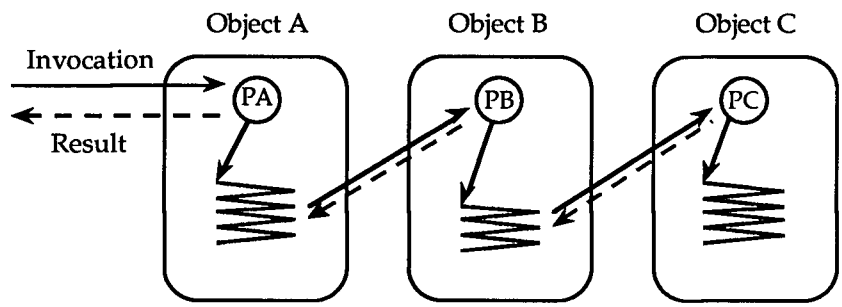

Figure 3.1: Performing an action in the active object model

In the *static* variation of the active object model, when an object is made operational, a fixed number of server processes are created for it. In the simplest case, a single process is created for each object. As a result, an object is restricted to handling at most one invocation request at a time. Requests which are delivered to an object while its process is busy are placed in a queue of pending requests to be subsequently processed or rejected. The complexity of a DOBPS is greatly reduced

when only a single process is permitted within each object. For example, a synchronization scheme (see §4.2.) does not have to be provided. However, this approach is also very restrictive since object concurrency is not permitted. An alternate approach is to create multiple server processes for each object. In this approach, when a request is delivered to an object, it is randomly assigned to an idle server process. Requests which are delivered while all the processes are busy are placed in the queue or rejected. This permits an object to perform multiple invocations concurrently, limited by the number of server processes that are created for the object.

In the *dynamic* variation of the active object model, server processes are dynamically created for an object as required. Requests are never queued, as is the case in the static variation of this model. When a request is delivered to an object, a new process is created for the object to handle the request. When the execution of the operation terminates, the process is destroyed. This approach permits an object to perform almost any number of invocations concurrently, limited only by the number of processes that can be created in the system. A second advantage of this approach is that an object can make use of as many processors which are available at a workstation as possible. In contrast, an object in the static variation of this model will only make use of as many processors as there are server processes. If there are fewer processes than processors, some processors will sit idle. A disadvantage of the dynamic variation of this model is the expense of dynamic process creation and destruction. Fortunately, this problem can be alleviated by the use of an additional mechanism, such as maintaining a pool of idle processes.

The major problem of the active object model is that deadlock may occur if an object does not have enough server processes to handle its requests. For example, if an object which has n server processes is invoked by the same action repeatedly (a nested invocation of over n times), the object will not be able to handle the (n+1)st request and as a result, deadlock will occur. Therefore, any action which makes more nested invocations on an object than it has server processes can never be

processed. Deadlock is obviously less of a problem in the dynamic variation of the model; but, it can still occur.

x When the active object model is supported, the large, medium and fine-grain object model (§3.1.) should not be used because of the great number of processes which will have to be maintained. Furthermore, when the active object model is supported, the direct invocation scheme (§5.1.2.) cannot be used due to the incompatibility of the two schemes.

√ The message passing scheme (§5.1.1.) should be used when the active object model is supported.

3.2.2. Passive Object Model

In the *passive object model*, the processes and objects of a DOBPS are completely separate entities. Typically, each process and each object resides in its own address space. A process is not bound to 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 during its lifetime. When a process makes an invocation on another object, its execution in the object which it currently resides is temporarily suspended. Conceptually, the process is then dynamically bound to, or mapped into the address space of, the second object where it executes the appropriate operation. When the process completes the operation, it is returned to the first object where it resumes its execution of the original operation at the point just after the invocation. (See Figure 3.2). A detailed description of this procedure is given later in this thesis (see §5.1.2.).

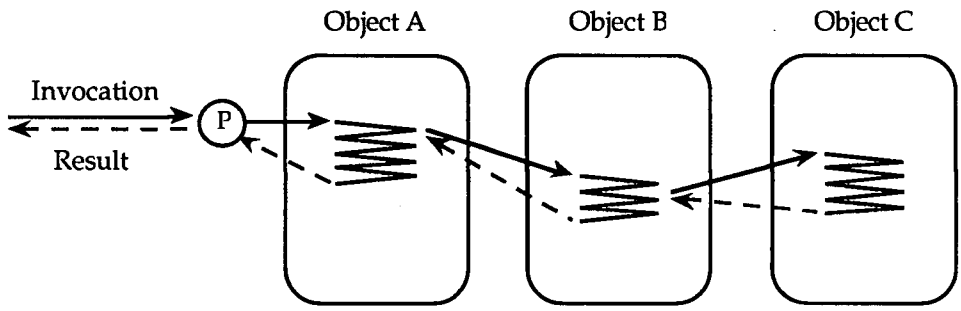

Figure 3.2: Performing an action in the passive object model

There are at least four advantages for using the passive object model. First, any number of processes can be bound to an object. Therefore, the amount of concurrency that can be handled by an object is limited only by the number of processes which can be created in the system. Second, deadlock is not a problem in the passive object model since each process corresponds to exactly one action, and vice versa. Third, the number of processes required at any one time is small compared to the active object model, in which at least one process is created for each object. Finally, it permits interactions between objects which reside on the same workstation to be performed relatively efficiently since an invocation is similar to a procedure call.

A major disadvantage of the passive object model is that the cost of mapping a process into the address space of an object is typically a difficult and an expensive operation. Another problem is that object migration (see §6.2.2.) is complicated because of the difficultly of moving the processes that execute within an object which is moved. If an object is moved, each process which is currently suspended in the object must be altered accordingly to ensure that the process properly returns when it completes the invocation.

x When the passive object model is supported, the pure message passing scheme (§5.1.1.) should not be used because the inefficiency of message passing eliminates the advantage of efficient intra-workstation object interactions.

√ The direct invocation scheme (§5.1.2.) should be used when the passive object model is supported.

3.3. An Overview of Object Structures in Existing Systems

Amoeba

Amoeba supports large-grain objects (or *Clusters*) and the static variation of the active object model which permits multiple server processes. Large-grain objects are composed of a number of code and data segments which may be shared by multiple objects.

Argus

Argus supports large-grain (*Guardians*) and medium-grain objects (*data 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 destroyed, it is returned to the pool. Only when the pool is empty are new processes created. The contents of the pool is cleared periodically by a garbage collector to ensure that memory is not tied up unnecessarily.

CHORUS

CHORUS supports large-grain objects (or *actors*) and the static variation of the active object model. It differs from most DOBPSs in that there is only a single server process per object and a server process cannot be interrupted or blocked while it is executing an operation. All invocations that the process makes while it is executing an operation are recorded in a transmit queue and then issued when the operation completes. (See Figure 3.3). CHORUS also differs from most DOBPSs in that a result may not be returned when an invocation completes.

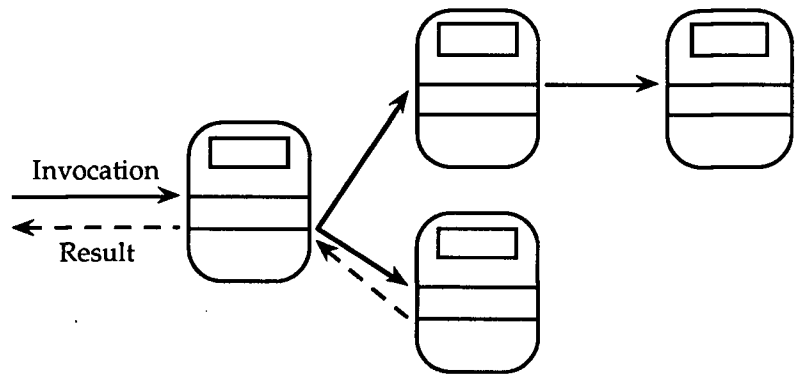

Figure 3.3: A CHORUS action

Clouds

Clouds supports large-grain objects and the passive object model. An object consist of a number of segments, including: a code segment, at least one data segment for persistent data, and a number of heap segments for volatile data. Clouds allows code segments to be shared by multiple objects. A process consists of a process control block and a stack which resides in its own address space. When a process makes an invocation on an object, the process enters the object through the entry points of one of its operations in its code segment. It performs the corresponding operation, and then leaves the object. Objects are efficiently mapped into the address spaces of processes by the use of paging hardware (see §6.1.).

Eden

Eden supports large-grain objects (or *Ejects*) and the static variation of the active object model. Each object has three components: a long term state which contains the persistent information, a short term state which contains the volatile information, and the code of the operations. An object also contains a number of processes, including: a Dispatcher process, at least one server process, possibly some maintenance processes, and a process which makes the object

dormant after it has been idle for a period of time. The Dispatcher process is responsible for accepting invocation requests delivered to its object and assigning them to idle server processes.

Emerald

Emerald supports large-grain (*global objects*), medium-grain (*local objects*), fine-grain objects (*direct objects*), and the passive object model. 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. The three types of objects have three different representations and characteristics. Large-grain objects may be invoked by any object of the system and they are the smallest entities that can be moved independently about the system (see §6.2.). Medium-grain objects are completely contained within other objects and each may be invoked only by objects which are also contained in the same object. They are moved along with the objects in which they reside. Fine-grain objects reside within large-grain or medium-grain objects. Each is used directly by the object in which it resides.

Emerald is different from most DOBPSs in that the code of an object is stored in a separate, immutable object referred to as a *concrete type object*. A concrete type object be shared by a number of objects which reside on the same workstation.

A process consists of a stack composed of a collection of *activation records*. Each activation record maintains information regarding the status of the process within an object it has invoked and is used while it is executing in that object. (See Figure 3.4).

Emerald also differs from most DOBPSs in that a single, common address space is shared by all objects and processes which reside in the same workstation. This lowers the cost of creating and maintaining objects. It also enables the objects which reside in the same workstation to examine and modify each other directly, thus permitting object interactions to be performed efficiently since

context switching does not have to take place. Using a single address space, however, has the drawback that hardware protection between objects which reside on the same workstation is not provided. Therefore, an object failure (see §4.4.) can be fatal to an entire workstation. Furthermore, the security of the objects may be compromised because objects are accessed directly.

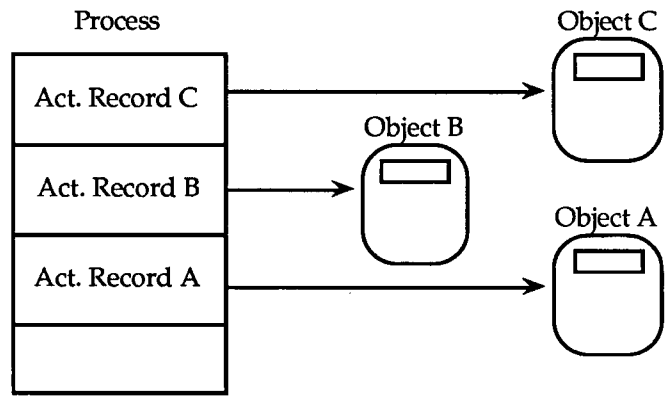

Figure 3.4: An Emerald process

TABS/Camelot

TABS and Camelot support large-grain (*Data servers*) and medium-grain objects (*data objects*), and the static variation of the active object model which permits multiple processes. They differs from most DOBPSs in that medium-grain objects are encapsulated by a number of server processes. These server processes accept invocation requests which specify the medium-grain object and the operation to be invoked.

Chapter 4.
Object Management

Objects are the fundamental resources of the DOBPSs, therefore their management is an essential function of these systems. In this chapter, we describe the features provided by DOBPSs for making the effects of an action on persistent objects permanent, for synchronizing the execution of multiple concurrent invocations within an object, and for protecting objects from unauthorized clients. We then describe features for recovering failed objects and for replicating objects to provide the property of availability. Finally, we present a brief overview of the manner in which objects are managed in a number of existing systems.

4.1. 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 which execute concurrently will be scheduled in such a way that the overall effect is as if they were executed sequentially in some order (see §4.2.).
- *Atomicity.* An action either successfully completes or it has no effect.
- *Permanence.* The effects of an action which successfully completes are not lost, except in the event of a catastrophic failure.

A single action may cause a number of related invocations to be made, possibly affecting multiple objects. The invocations of an action may be performed sequentially or concurrently, and they may be either essential or non-essential to the outcome of the action. When all the invocations associated with an action complete, the corresponding action successfully completes. The modifications made to persistent objects by the action should then be made permanent by immediately recording the changes to secondary storage (see §6.1.2.). An action which successfully completes is said to *commit,* while an action which fails to do so is said to *abort* and all its modifications are undone. A DOBPS may permit either the user or the system to be responsible for initiating the procedure for committing an action (see §4.1.3.).

4.1.1. User Managed

In the *request* scheme, the user is entirely responsible for handling invocation failures (see §5.3.1.) and for initiating the procedure to commit an action. This scheme has the added flexibility of permitting the user to decide whether or not the effects of an action are to be permanent when the action completes. There is no requirement that every action be committed. The advantage of not committing every action is that the performance of the DOBPS will improve since executing the commit procedure is a relatively expensive task. The problem of not committing every action is that the guarantee of permanence, and possibly atomicity, is lost. For example, if only some of the changes made to objects by a completed action are recorded, then some of the effects

of the action will be lost. Consequently, a workstation could be restored to an inconsistent global state (see §4.4.1.) when the request scheme is supported.

4.1.2. System Managed

In order to guarantee that the properties of an action will always hold, the system has to be entirely responsible for managing the actions and committing them when they complete. There are at least two schemes which can be used: the transaction scheme and the nested transaction scheme.

In the *transaction* scheme [Gray 80], the invocations are essential to the outcome of an action. The failure of an invocation (see §5.3.1.) causes the corresponding action to abort. If all the invocations associated with an action successfully complete, the system performs the commit procedure to make their modifications permanent. Only when the commit procedure successfully completes, does the action commit. If the commit procedure fails, the action aborts. The transaction scheme is a simple, straightforward scheme which ensures that either all the modifications made by an action are completed and committed, or none are. However, there are two drawbacks with this scheme: the overhead of the commit procedure occurs every time an action completes, and the failure of any invocation causes the entire action to fail.

An extension of the transaction scheme is the *nested transaction* scheme [Moss 85] in which an invocation can be non-essential to the outcome of its action. That is, the failure of an invocation may or may not cause the corresponding action to abort. In this scheme, a single *top-level action* creates and manages multiple lower-level, autonomous *sub-actions* which execute concurrently. The failure of a sub-action does not force the top-level action to fail. Instead, the top-level action can handle the failure in any way it chooses. For example, it may choose to perform the sub-action again or it may simply ignore the problem. The modifications made to objects by sub-actions which successfully complete are conditional to the success of their top-level action. Only when a top-

level action commits are the modifications made by its sub-actions made permanent. If a top-level action aborts, all the changes made by its sub-actions are 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. A drawback of the nested transaction scheme is that the representation of objects in memory is complicated (see §6.1.1.).

4.1.3. The Commit Procedure

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 78].

(a) A *pre-commit* request is issued to the affected objects.

(b) An object which receives a pre-commit request writes its modifications to secondary storage and returns an acknowledgement.

(c) If any object fails to respond to a number of pre-commit requests, an *abort* request is issued to the affected objects and the commit procedure ends. If all return acknowledgements, a *commit* request is issued to the affected objects.

(d) An object which 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 acknowledgement. An object which receives an abort request discards the changes and resets the control structures.

(e) A commit request is repeatedly issued to the objects which fail to respond until all return acknowledgements, at which point the commit procedure ends.

This commit protocol is simple and efficient. However, a problem with the standard two-phase commit protocol is it may block when the coordinator (see below) fails or when a network failure occurs, until the failure is fixed [Spector 87a]. Therefore, other commit protocols may be used.

A DOBPS can support either a single coordinator approach or a fan-out approach for coordinating the commit procedure when an action completes. In the single coordinator approach, one object is informed of all objects which are affected by an action. This object is then responsible for performing the commit procedure on each of the objects. The advantage of this approach is its relative efficiency because the commit procedure is performed directly on all objects affected by an action, instead of indirectly as is the case in the fan-out approach. The difficulty with this approach is collecting and propagating the required information to the coordinator.

In the fan-out approach, the responsibility for performing the commit procedure is distributed amongst a number of objects. There are two variations of the fan-out approach. In the first variation, when an action completes, the object that initiated the action performs the commit procedure on those objects that it has invoked. These objects in turn, perform the procedure on those objects they invoked, and so on. A problem with this variation of the fan-out approach is that each object has to be responsible for handling part of the commit procedure and for maintaining a record of the objects it invokes. The advantage of this variation is that it does not have the problem of complicating object mobility as the second variation does. In the second variation of the fan-out approach, each operating system is responsible for maintaining a record of the objects that are invoked by an action, and for performing the commit procedure on those objects when the action commits. The drawback of this variation is that object mobility is complicated because machine-dependent information is maintained. A problem with the fan-out approach in general is considerable delays can occur when the commit procedure has to propagate through a large chain of invocations.

4.2. Synchronization

Another important function of a DOBPS is ensuring that the activities of multiple actions, which invoke the same object simultaneously, do not conflict or interfere with each other. An action should never be able to observe or modify the state of an object which has been partially modified by another action that has not committed. Failure to enforce this rule may lead to *cascading aborts*. That is, any action which observes the partially modified state of an object created by an action which later aborts, also has to be aborted. Therefore, 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, some are surveyed by Benstein and Goodman [Benstein 81]. These synchronization schemes can be separated into two classifications: pessimistic schemes and optimistic schemes.

In a *pessimistic* synchronization scheme, appropriate steps are taken to prevent conflicts from occurring. Any action which invokes an object is suspended if it will interfere with another action which is currently being processed by the object. When all conflicting actions commit, a suspended action is resumed. Locks are the most common mechanisms used by a pessimistic synchronization scheme; however, timestamps, semaphores and monitors are also used. When a locking mechanism is used, an action first has to acquire the appropriate lock before it can invoke an operation of an object. If it can not acquire the lock, it blocks until one is available. For example, each operation of an object can be defined as requiring either a read lock or a write lock and the following locking rules can be enforced: an action can acquire a read lock only if no other action holds a write lock, and a write lock can be obtained only if no other action holds a read or a write lock[7]. Once a lock is acquired, it is held until the action commits or aborts. Locking is a simple,

[7] Furthermore, an action can acquire a read lock if and only if all holders of write lock are ancestors of the same nested transaction; and a write lock can be obtained if and only if all holders of read and write locks are ancestors [Liskov 87].

effective, and efficient scheme. However, it also introduces the possibility that two or more actions may deadlock, each action attempting to acquire a lock held by the other. To eliminate this problem, a deadlock detection mechanism such as a watchdog timer or a deadlock avoidance graph is needed.

x[8] When a pessimistic synchronization scheme is supported, objects should not be represented in memory using a multiple versions approach (§6.1.1.) since ensuring synchronization is made more difficult when control information is maintained at multiple locations.

√ Consequently, objects should be represented in memory using a single version approach (§6.1.1.) when a pessimistic synchronization scheme is supported

In an *optimistic* synchronization scheme, steps are not taken to prevent conflicts from occurring while invocations are being processed. Each action is typically given its own version of the object on which to make its modifications (see §6.1.1.) and multiple actions are permitted to execute concurrently. However, before an action can commit, it is tested for serializability to ensure that the information it observed does not conflict with changes made by another action which has recently committed. That is, the system determines if the data which was examined by an action in its version of the object is still up-to-date, or whether it has since been altered. If the data has not changed, then the action can commit. Otherwise, the action will have examined some information which is now out of date and consequently its modifications based on that information will be incorrect. Consequently, the action is aborted. To reduce the likelihood of conflicts, objects should be represented either as a number of pages (see §6.1.) or as a number of medium-grain and/or fine-grain objects (see §3.1.). The main advantage of the optimistic synchronization scheme is that it permits the maximum degree of concurrency possible within an object. Actions never have to wait

8 The definition of the symbols "x" and "√" can be found in §1.3. on page 6.

to be performed, as they do in the pessimistic synchronization scheme. Unfortunately, this scheme may also force some actions to abort and have to be redone. As a result, multiple copies of each object or an undo/redo log (see §6.1.2.2.) have to be maintained in case the modifications need to be undone.

x When the optimistic synchronization scheme is supported, objects should not be represented in memory using the single version approach (§6.1.1.) nor should they be represented in secondary storage using the pure checkpoint scheme (§6.1.2.1.) because of the difficulty of integrating these schemes.

√ Objects should be represented in memory using the multiple versions approach (§6.1.1.) when the optimistic synchronization scheme is supported. Furthermore, objects should be represented in secondary storage using the history of checkpoints scheme (§6.1.2.1.) or the redo log scheme (see §6.1.2.2.) to enable the commit procedure to determine the changes that have been made by each committed action.

A pessimistic scheme avoids the overhead of undoing and redoing requests at the expense of reduced concurrency. For example, two actions which 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, while the reverse holds true when conflicts are infrequent. A detailed discussion of the pessimistic and optimistic synchronization schemes is given in [Herlihy 86] and [Sheth 86].

4.3. Security

Preventing unauthorized clients from successfully invoking an object is a very important but often ignored function which should be provided by a DOBPS. There are typically two levels of security: first, a client must have knowledge of an object before it can contact the object; second, a client must present some authorization indicating it has permission to make an invocation on an object. The first level of security is inherent to all systems, but it is also very limited since the security has to be handled explicitly by the users of the system. This is typically done by not informing other users of confidential objects. To provide the second level of security, a special protection mechanism is required. A DOBPS permits either the system or the user to be responsible for supplying a security mechanism.

4.3.1. Capabilities

A common security mechanism is the *capability* scheme, which incorporates protection into the naming scheme [Mullender 86, Tanenbaum 86]. A capability consists of two fields: a name field and an access rights field. The name field specifies the object being invoked. The access rights field indicates the specific operations the possessor of the capability may invoke. Each capability refers to exactly one object, while multiple capabilities may refer to a single object. This enables the owner of an object to vary the access rights field for different clients. Conceptually, a capability is a key to a particular object which may be passed among the clients of a system. In order for a client to make an invocation on an object, it first has to possess one of the object's capabilities. When the capability is obtained, it is simply passed as a parameter in the invocation request. The name field is used by the system to determine where to deliver the request. The access rights field is used to determine if the client has the proper authority to invoke the specified operation. Validating the access rights field is usually the responsibility of the objects;

however, the system may perform this task. If the client does not have the proper access rights, then the request is ignored or rejected. A major problem with the capability scheme is that the system has to assume that its users can be trusted not to disclose the capabilities to unauthorized clients. For it is very difficult, and sometimes impossible, to retract a capability once it has been disclosed. To alleviate this problem, objects may supply their own security schemes, such as a password scheme, in addition to the capability scheme.

In the *system-managed* version of the capability scheme, each operating system is responsible for managing the capabilities. The capabilities possessed by an object are usually maintained in the kernel space of the workstation on which the object resides, and are typically stored in a separate, system managed list of capabilities [Cohen 75]. An object is not permitted to examine or modify the capabilities it possesses directly. Instead, to protect capabilities from being altered by the objects which possess them, they are always referenced via their index number in the objects' lists. For example, when a new capability is passed to an object, the system creates a new entry in the object's list and replaces the capability with the corresponding index number. There are at least three drawback of the system-managed version of the capability scheme. The first problem is the additional performance overhead. In order to examine the access rights field of a capability or to pass a capability to another object, a system call has to be performed. Objects cannot perform these activities directly. The second problem is object mobility (see §6.2.) is complicated because the operating system maintains capability information which must be moved when an object moves. Finally, the problem of garbage collection is introduced. The last problem occurs when a capability is stored on a workstation but all references to it lie elsewhere.

Another problem with the system-managed version of the capability scheme is that it is far from tamper-proof. This approach was originally designed for centralized systems, such as Hydra, in which the operating system resides in a single machine and cannot be tampered with. A DOBPS

on the other hand, distributes the operating system on a collection of autonomous workstations. Therefore, a malicious user may be able to alter the operating system of his workstation to bypass the security mechanism. For example, a user may have his operating system modify the access rights field of any capability he possesses. This breach in security is simplified because most DOBPSs which use a system-managed capability scheme typically do not provide any other security mechanisms such as encrypting the capabilities. These DOBPSs usually make the assumption that all workstations of the network can be trusted.

In the *user-managed* version of the capability scheme, each object is responsible for managing and maintaining the capabilities it possesses. The system, however, takes precautions to make modifying or forging a capability very difficult. The most popular method of protection is encrypting the capabilities so that only the object which a capability references can correctly observe the access rights field. The user-managed version of the capability scheme permits capabilities to be examined and passed efficiently. It also eliminates the problems of an operating system being altered to bypass the security mechanism, and object mobility since the capability information is maintained within the objects. The drawback of this scheme is the additional expense of protecting the capabilities such as the cost of their encryption and decryption.

In the *system & user-managed* version of the capability scheme both the operating systems and the objects are responsible for managing and maintaining the capabilities. Each object maintains a copy of the capabilities it possesses. Each operating system maintains a list of client-capability pairs that indicates which clients possess capabilities to objects which reside on the workstation, and the specific capabilities each client possesses. Whenever an invocation request is received, the operating system consults its capability list and verifies that the client which issued the request owns the particular capability used. An invocation request which contains an altered capability is destroyed or rejected. The system and user-managed version of the capability scheme

permits capabilities to be examined efficiently while ensuring they are not modified by a client. In addition, this scheme has the advantage that an operating system cannot be altered to bypass the protection scheme. There are a number of disadvantages with this scheme. The first drawback is the overheads of the dual maintenance and the verification procedure. The second problem is that the capabilities cannot be passed about the system as freely as in the other versions of the capability scheme. A capability can be obtained only from the object which it references, it cannot be obtained from other clients which already possesses a capability to the object. This reduces the flexibility of the scheme, but it also increases the security. The final problem is that object mobility is complicated because each operating system maintains capability information which must be moved when an object moves.

4.3.2. Control Procedures

Another type of security mechanism is the *control procedure* scheme [Banino 82]. In this scheme, each object has two procedures: one to check the incoming requests and another to check the outgoing results. All incoming invocation requests have to pass through the first procedure which checks their authorization and destroys all invalid requests. Similarly, all results pass through the second procedure. These procedures can support any type of security scheme. For example, they can use a scheme which ensures that a request is not received from or a result is not delivered to an unauthorized client, or they can use a password scheme to block unauthorized requests. The control procedure scheme permits each object to provide security tailored to the requirements of the object. For example, if an object is unimportant, then a minimal security scheme can be used; if an object is confidential, then multiple schemes can be used. Bypassing the security mechanism is extremely difficult because each object is entirely responsible for enforcing its own protection scheme.

4.4. Object Reliability

A DOBPS must be able to detect and recover from object and workstation[9] failures. This is an important feature which has to be provided because as the number of components in a system increases, the chance that a failure will occur also increases. An essential property for confining the effects of a failure and isolating it from the rest of the system is modularity. This enables autonomous entities to be created. These entities serve well as the units for recovery because they can be restored independently. Two types of modules are provided by DOBPSs: objects and workstations.

A DOBPS has the property of *fault tolerance* if it can survive failures, usually at a cost of reduced functionality and/or performance. Only the entity which has failed is lost, the rest of the system can continue functioning. On the other hand, a DOBPS has the property of *availability* if all the services and functions of the system are accessible all the time. If any service becomes inaccessible, the entire system is shut down and restarted, thereby restoring full service. There are at least two methods of providing object reliability despite object and workstation failures. The first method is to recover a failed object as quickly as possible, limiting the amount of time it remains unavailable. The second method is to replicate an object at multiple workstations so that it has a high probability of surviving workstation failures.

4.4.1. Object Recovery

An object failure can be caused either by a software error, such as a division by zero, or by a hardware error, such as the workstation on which it resides fails. The failure of an object or a workstation can be signalled by an invocation failure (see §5.3.), or each operating system can

---

[9] In most systems, network failures are indistinguishable from machine failures, therefore they will not be considered separately at this time. A further discussion on network failures is given in §8.2.

periodically probe its counterparts (see §5.3.1.) in the network to detect these failures. When the failure of a persistent object is detected, the system is responsible for restoring the object to a consistent state. When a workstation failure is detected, the system may be responsible for restarting the workstation and then recovering all the objects which reside on it.[10] The procedure for restoring the state of an object may be very simple or very complex. This depends on the representation of the objects in secondary storage (see §6.1.2.) and the type of recovery scheme supported: a roll-back scheme or a roll-forward scheme.

4.4.1.1. Roll-back Schemes

In a *roll-back* recovery scheme, a failed object is restored to its last consistent state which was recorded in secondary storage (see §6.1.2.). All processes and invocations that were being performed when the failure occurred are lost. In addition, all modifications that were made to the object by an action which did not commit are undone. Depending on how objects are represented in secondary storage, restoring an object to its last consistent state may require a DOBPS to provide a special recovery procedure. The roll-back recovery scheme is a very simple and efficient scheme which guarantees that a failed object is restored to a consistent state.

4.4.1.1.1. Checkpoint Recovery

A scheme used by traditional systems to enable them to recover failed processes is to periodically record the entire state of a process to secondary storage. This is referred to as a *checkpoint*. Consequently, a process which fails is restored to the state of some previously recorded checkpoint. The main problem with this scheme is obtaining a checkpoint which records a consistent state of a process, and a set of checkpoints which record a consistent *global state* of a

[10] To simplify the discussion of object recovery, we ignore the problem of workstation recovery.

system. A process that does not interact with any other process is always in a consistent state and therefore can be checkpointed at any time. A group of processes that interact on the other hand, must be checkpointed so that the individual checkpoints are consistent with one another with respect to their interactions. That is, they have to be checkpointed while they are not interacting. If a client process is checkpointed before an interaction and the server after the interaction, then a consistent state of the two processes will not have been recorded. The client will not have a record of sending a message, but the server will have received a message. As a result, these two checkpoints cannot be used to restore a system to a consistent global state.

Figure 4.1 shows two globally consistent states, checkpoints 1 and 2, and one globally inconsistent state, checkpoint 3. The processes should not be restored to the states of the third checkpoint since process c will have observed the interaction, but process b will not have. Therefore, the processes should be restored to the states of the second group of checkpoints.

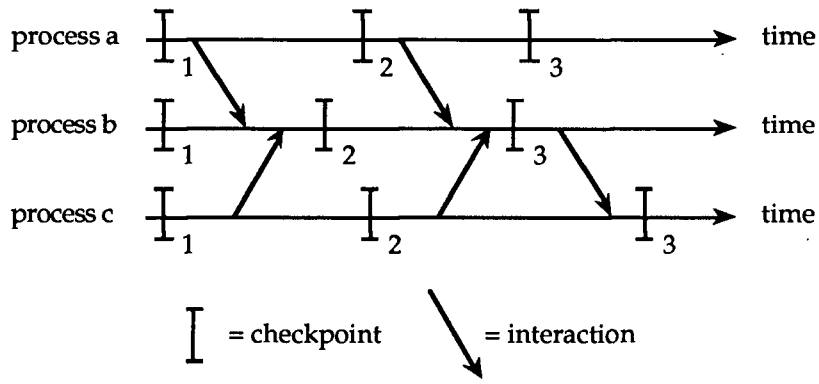

Figure 4.1: Checkpointing processes and global state.

The major problem with this scheme is that the failure of a process may cause extensive amounts of work to be lost in order to restore the processes it interacts with to their last consistent states. This is referred to as the *domino effect*. Figure 4.2 shows an example where the only global

state which is consistent is the first checkpoint taken by the two processes. Therefore, when the failure occurs, the two processes will have to be restored to these states.

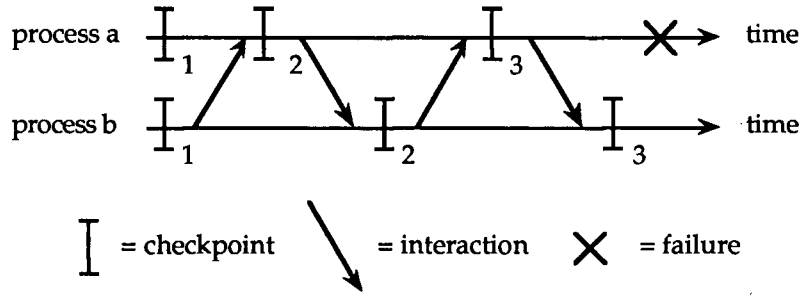

Figure 4.2: The domino effect.

DOBPSs provides features to alleviate this problem. First, objects serve well as units to checkpoint. Second, the completion of an action indicates a point in time when a group of objects have completed their interactions and are therefore in a consistent state.

A DOBPS which records its objects in secondary storage using a checkpoint scheme (see §6.1.2.1.) does not require a recovery procedure to restore its failed objects. Instead, recovery is done automatically by the memory and secondary storage features of the system (see §6.1.). That is, the next invocation on a failed object will cause a copy of its most recent checkpoint to be loaded into memory for execution; this procedure also completes the recovery.

4.4.1.1.2. Log Recovery

A DOBPS which records its objects using a log scheme (see §6.1.2.2.) requires a recovery procedure to restore its failed objects. The recovery procedure typically loads a base checkpoint of a failed object into memory and then the log entries are used to restore the object to its last consistent state. The problem with using a recovery procedure is it adds overhead to a system

whenever an object is restored. Furthermore, the larger the log, the larger the overhead. As a result, the advantage of using a log to reduce the expense of recording the modifications made to objects is partially offset by an increased cost of recovery. Details of the recovery procedure for the various types of logs are described below.

A redo log records enough information to allow the modifications made by an action on an object to be redone. For example, it may record the new values of the modified pages of objects. During the recovery procedure, the entries in the log associated with the failed object are examined. Those entries corresponding to an action which has committed are used to re-perform the modifications on the object's base checkpoint in the same order they were originally performed.

Similarly, an undo/redo log records enough information to allow the modifications made by an action on an object to be completely undone or redone. For example, it may record both the old and the new values of the modified pages of objects. During the recovery procedure, those entries of the log associated with the failed object are examined. The modifications which correspond to actions that have committed are re-performed, while those which correspond to actions that have aborted are undone. This restores the object's base checkpoint to its last consistent state.

4.4.1.2. Roll-forward Schemes

In a *roll-forward* recovery scheme, a failed object is restored to the state just before the failure occurred. All processes and invocations that were in progress at the time of the failure are restarted and allowed to complete. A roll-forward recovery procedure is much more complex than a roll-back recovery scheme because the system makes it appear that the failure did not occur.

A commit log (see §6.1.2.2.) records enough information for an object which is associated with an action that was being committed when a failure occurred to be restored and the commit procedure

completed. This is actually only a partial roll-forward recovery scheme since it does not allow the modifications made to an object to be restored if the failure occurs before the corresponding action attempts to commit. During the recovery procedure, the entries of the commit log are examined in the reverse order in which they were entered to find the most recent data entry (checkpoint) corresponding to the object. The stage of the commit procedure reached by the action which is associated with that checkpoint is then determined by examining the log again to find the most recent outcome entry (status) for the action. The following algorithm is then used.

(a) If the outcome entry indicates the action committed, then the object is restored to the state of the data entry.

(b) If the outcome entry indicates the action aborted or there is no corresponding outcome entry for the action, then the data entry is ignored and the previous data entry of the object is examined. The recovery procedure is then repeated.

(c) If the outcome entry indicates the action pre-committed, then the object is restored to the state of the data entry and the commit procedure for the action is restarted when the recovery procedure completes.

A commit log enables the system to recover modified objects which were in the process of being made permanent when the failure occurred. Unfortunately, the commit log adds considerable overhead for this extra feature.

A pure roll-forward recovery scheme permits all the modifications made to a failed object to be restored. One variation of this scheme requires a log to record all the messages, or interactions, between the objects[11]. When an object fails, its base checkpoint is used as the initial state of the new object in the recovery procedure. The log is then examined and all the messages originally sent to the object between the time of the checkpoint and the time of the failure are resent to the new

[11] This approach is derived from Powell and Presotto Publishing technique [Powell 83].

object in their original order. Messages sent by the new object during this recovery period are discarded. Once the new object reaches the state just before the failure occurred, it is allowed to proceed on its own. This variation of the roll-forward scheme is very effective at restoring the state of an object to the point just before the failure. There are a number of problems with this scheme, however. The major problem is that the log has to reliably record all of the interactions between all of the objects all of the time. If it fails to record a message, the entire recovery procedure may fail. A second problem is handling requests and results that are delivered to an object during the recovery period. The object cannot be allowed to observe these requests because it would interfere with the recovery procedure; however, they should not be ignored either. Another problem, associated with all roll-forward recovery schemes, is an object failure which is caused by a software error will occur again when the object is recovered. This will cause recovery to take place again and the failure to occur again, and so on. Roll-forward recovery schemes are generally not used because of problems such as these.

- x When a roll-forward recovery scheme is supported, the objects cannot be represented on secondary storage using a checkpoint scheme (§6.1.2.1.) because additional information is required by the recovery procedure.
- √ Objects should be represented on secondary storage using a log scheme (§6.1.2.2.) when a roll-forward recovery scheme is supported.

4.4.2. Object Replication

In order for important objects to survive workstation failures, a DOBPS may permit them to be replicated at different workstations to increase their availability, to a high probability. Consequently, the failure of a workstation in which a replica of an object resides only results in the unavailability of that replica. A replication scheme enables a DOBPS to tolerate a number of workstation failures while still allowing it to provide full functionality. It can also enhance the

performance of a system by permitting a replica to be created at a workstation which has objects that interact with the object frequently, thereby reducing the overhead of their invocations. Unfortunately, there are a number of problems associated with replicating objects, including: ensuring that the states of the replicas are consistent, and synchronizing the activities of multiple clients. Furthermore, a network partition can create havoc with a replication scheme (see §8.2.).

There are two types of availability: write availability and read availability. *Write availability* means as long as a client is able to access a replica of an object it may modify or examine the object. *Read availability* means as long as a client is able to access a replica it may examine the object.

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. Therefore, they can be replicated without the problems of maintaining state consistency and synchronization. A deficiency of this scheme is that it is very limiting since it only provides additional read availability.

An alternate approach is the *primary copy* replication scheme [Alsberg 76]. In this scheme, one of the replicas of an object is designated as the primary copy while an ordered collection of replicas are maintained on separate workstations as secondary copies. A read request can be performed by any replica. A write request on the other hand, is first performed by the primary copy which then propagates the modifications to 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, the failure of a primary copy results in the object not being able to perform write requests until the primary copy is recovered. Therefore, it provides additional read availability, but not write availability. The *dynamic* primary copy scheme on the other hand, provides additional write availability. The failure of a primary copy results in one of its secondary copies taking over as the new primary. The failure of a secondary copy results in it being

restored to an up-to-date state when it is recovered. Unfortunately, there are a number of problems associated with the dynamic primary copy replication scheme in addition to the problems of replication previously mentioned. First, when a secondary copy replaces a primary copy, it has to take over the identity and the resources, such as the ports or locks, of the failed object so that its clients do not perceive any change in the server object. Second, the replicas are usually fixed to specific locations which cannot be easily changed. This can lead to problems if workstations are down for long periods of time. Finally, a network partition which separates a primary copy from some of its secondary copies causes two sets of objects to be created.

The *peer objects* replication scheme is another approach. In this scheme, there is no designated primary object or secondary objects. Instead, every replica is considered to be equal. A read request or a write request can be performed by any replica; however, it may require the cooperation of some or of all the other replicas. There are a number of variations of the peer objects scheme, including: Voting, Available Copies, and Regeneration. A detailed comparison of these three schemes is given in [Pu 86].

The *voting* scheme [Gifford 79] requires the cooperation of a pre-determined number of replicas in order to perform a read or write request. This scheme enables the failure of a limited number of replicas to be tolerated while ensuring that consistency is maintained if a network partition occurs. The main drawback of the voting scheme is that the replicas of an object are fixed to specific locations; therefore, a failed replica remains unavailable until its workstation is restored.

The *available copies* scheme [Bernstein 84] requires one replica to perform a read request and all available replicas to perform a write request. When a workstation failure is detected, the rest of the system is notified of the unavailability of those replicas which reside on that workstation. When the workstation is recovered, new up-to-date replicas are obtained and the rest of the system

is notified of the recovery. A disadvantage of this scheme is the overhead of notification has to be endured every time a workstation fails or recovers. The available copies scheme also fixes the replicas of an object to specific locations.

The *regeneration* scheme [Pu 86] is similar to the available copies scheme except that the replicas are not fixed to particular locations. A read request is sent to successive replicas until one services the request. A write request is sent to all the replicas to service the request. If some of the replicas are unavailable, new up-to-date ones are created on other workstations to replace them and the request is reissued. When a failed workstation is restored, any replicas which reside on the workstation are destroyed because they will have been, or will be, replaced by the regeneration mechanism. This helps remove inconsistencies in the system. A main advantage of the regeneration scheme is that it can dynamically adapt to changes in the hardware environment. For as long as one replica of an object is available, the object can be examined or modified. In addition, the number of replicas needed for an object to survive workstation failures is smaller than the number needed by other replication schemes. A drawback of the regeneration scheme is a network partition which separates some replicas of an object results in the two sections of the network regenerating the object independently.

4.5. An Overview of Object Management in Existing Systems

Amoeba

Amoeba supports the request scheme, and both a pessimistic and an optimistic synchronization mechanism. The pessimistic locking scheme is used for actions which affect multiple objects while the optimistic scheme is used when only a single object is affected. In the optimistic scheme, when an action first modifies an object it is given a copy of the most recent version of the object on which to make its modifications. Before an action may commit, however, it

is tested to determine if the serializability constraint is satisfied. To reduce the chance of conflict, an object's state is divided into pages which can be updated independently. The following two tests are then 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, or current, version. If it is, the commit procedure is performed. Otherwise, a page-by-page test of the current version, the modified version, and the past version of the object is performed. The second test determines if the pages of the modified version examined by the action are the same in the current version as they were in the past version. If any of the pages have changed, the action is aborted. Otherwise, a new version of the object is created by combining the updates made to create the current version and the modified version, and then the commit procedure is performed.

This second test may be clarified with an example. (See Figure 4.3). Version X of an object is the current version when three actions (A, B and C) make modifications to it. Each is given its own copy on which to make its changes, and each commits independently. Action A reads from page a and writes to page b (creating b2). It commits first, creating version Y of the object. Action B also reads from page a, but writes to page c (creating c3). The pages read by action B (page a) do not conflict with those written by action A (page b), so action B can commit. The modifications made by both action A and B are then combined to create version Z of the object. Action C reads from pages a and b, and writes to page b. However, since one of the pages it read (page b) has been modified by another action (action A) which has already committed, action C cannot commit. Action C could have committed though if its version of the object was based on version Y, instead of version X, because it would have observed up-to-date information. A detailed description of this optimistic concurrency control mechanism is given in [Mullender 85].

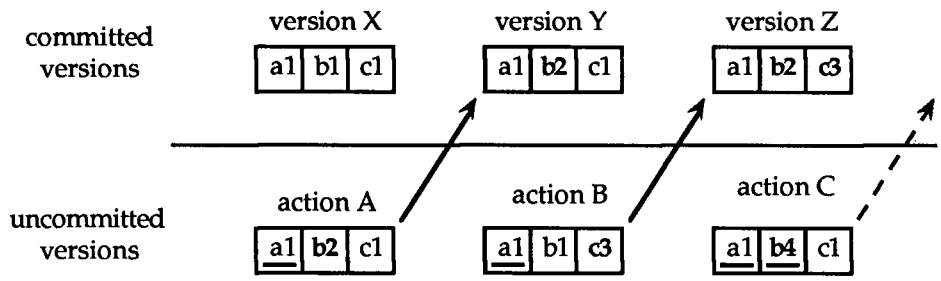

Figure 4.3: Optimistic synchronization in Amoeba

Security is provided using a user-managed capability scheme which may be augmented with additional protection mechanisms, such as: an encryption/decryption scheme or a hardware protection scheme. A random number generator which produces large, sparse numbers is used to inexpensively create object identifiers. This makes it difficult, though not impossible, for a user to forge a capability. In addition, the capabilities may be encrypted so that they cannot be altered easily or special hardware may be used to ensure that only the server object which is invoked actually receives the request. A detailed description of these security schemes is given in [Mullender 86].

A special boot server object is used to detect and recover objects and workstations which have failed. An object can register with the boot server to ensure that it is not down for a long period of time. Periodically, the boot server probes each of its registered objects to determine if they are still alive. Any object which does not respond to a number of probes sent to it is assumed to have failed and the recovery procedure is performed. If the object has failed but the workstation it was residing on is still working, then the object is restored on that workstation. If the workstation has failed, then it is restarted and the object is restored. If the workstation cannot be restarted, then the object is restored on another workstation. The roll-back recovery scheme which uses checkpoints is supported.

An object replication scheme is not provided.

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 and the large-grain object returns a result, its identifier is added to a list of all large-grain objects which were affected by this action. This list is eventually propagated back to the object which initiated the action to enable it to perform the commit procedure directly on all the large-grain objects affected. Each large-grain object, however, is responsible for performing the commit procedure on its own medium-grain objects.

A pessimistic synchronization scheme which uses read/write locks is supported. The locking rules are simplified by not allowing a top-level action to execute concurrently with its sub-actions. This ensures that only one action or sub-action can modify an object at a time. When a sub-action successfully completes, its locks are inherited by its top-level action. To prevent deadlock, a long time-out is used.

Argus does not provide a security scheme.

Each operating system provides a guardian manager which is responsible for creating new large-grain objects at the workstation, detecting objects which have failed, and recovering objects after a failure. A partial roll-forward recovery scheme which uses a commit log is supported.

An object replication scheme is not provided.

CHORUS

CHORUS supports the request scheme. A single action is able to create multiple sub-activities which execute concurrently. This scheme is similar to the nested transaction scheme, but

it is not as flexible or as powerful. In this scheme, a sub-activity is a completely separate entity which may commit independent of the top-level action.

A pessimistic synchronization scheme is supplied since each object can perform at most one invocation at a time.

The control procedure security scheme is supported with additional protection provided by the objects' ports (see §5.1.1.). When a port is created, its creator informs it which objects may open or destroy it. Only those objects which are authorized to open a particular port may do so.

The roll-back recovery scheme which uses checkpoints is supported.

For replicating objects, a variation of the primary copy scheme is supported. In this variation, there is a single secondary copy which acts solely as a backup and is otherwise not used. The primary copy and the backup copy consecutively perform the same invocations. The primary is responsible for accepting and processing invocation requests, and propagating completed requests to the backup. The backup is responsible for processing the invocation requests delivered to it, but suppressing the invocation requests it would normally issue when it completes. Therefore, the backup performs exactly the same operations as the primary, but it is delayed by one operation and it always records the last consistent state of the primary. The primary copy is also responsible for periodically checking the status of the backup copy, and vice versa. If the primary fails, then the backup detects this, takes its place, and creates a new backup object. Similarly, if the backup fails, then the primary detects this and creates a new backup copy to take its place.

Each operating system provides an inspector object which periodically checks with its counterparts in the network to detect workstation failures. In addition, if a primary copy or a backup copy detects the failure of the other, it notifies the local inspector object which then performs the appropriate recovery procedure.

Clouds

Clouds supports the nested transaction scheme. The system provides an action manager object which is responsible for maintaining a record of all objects visited by an action. This object is also responsible for coordinating the commit procedure for all objects modified by an action and making the appropriate invocations on the objects. Each persistent object has three default operations: pre-commit, commit, and abort. If the pre-commit operation is invoked, the object writes a checkpoint of its state to secondary storage. If the commit operation is invoked, the modifications are made permanent by making the checkpoint the new current version. If the abort operation is invoked, the modifications are discarded.

Two pessimistic synchronization scheme are provided: a custom scheme and an automatic scheme. The custom scheme allows a user to create his own synchronization rules using semaphores or locks. The automatic scheme uses the read/write locks with each operation defined as requiring either a read or write lock. Clouds cannot enforce the serializability property of actions because the users can define their own synchronization rules. This has the benefit of enhancing concurrency because an action may not have to wait for another action to complete before it can make an invocation on the object. However, it also has the problem of cascading aborts. Therefore, while it allows additional flexibility, it may also cause great inefficiencies.

A system-managed capability scheme is provided. Clouds is different from most systems which use this scheme in that the capabilities are maintained in the objects which possesses them. The objects are responsible for authenticating them; however, they can only be examined and passed through system calls.

The roll-back recovery scheme which uses checkpoints is supported.

Object replication is supported using the immutable objects scheme and the peer objects scheme. In the approach used by Clouds, all replicas of an object have the same user-level identifier. At the system level, a replica number is appended to the user-level identifier to create a unique identifier for each object. Therefore, a single user-level object identifier can refer to a set of objects. When a failed workstation is restarted, each replicated object which resides on the workstation is checked to determine if it is up-to-date. If it is found to be obsolete, then a copy of its most recent version is obtained.

Eden

Eden supports the nested transaction scheme, but the action has to be explicitly committed. The system does not automatically initiate the commit procedure when the action completes. A special manager object is created for each nested transaction to manage its activities. A 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 which uses monitors is supported.

A variation of system & user-managed capability scheme is provided. In this variation, each operating system records its own copies of the capabilities possessed by object which reside on its workstation. Whenever an invocation is made, the operating system of the client verifies the capability used by comparing it against the system's copy. An invocation request which contains an altered capability is destroyed, all others are allowed to proceed. A problem with this scheme is an operating system can be modified to bypass the security mechanism.

A roll-back recovery scheme which uses checkpoints is supported.

Object replication is supported using the immutable objects scheme and the regeneration version of the peer objects scheme. For each replica of an object, a version manager (see §6.4.) is created at the workstation on which the replica resides. The version managers of an object interact to ensure that inconsistencies do not arise among the replicas of an object.

Emerald

Emerald supports the request scheme and a pessimistic synchronization scheme which uses monitors.

Emerald assumes that all workstations can be trusted; therefore, it does not provide a security scheme.

A roll-back recovery scheme which uses checkpoints and the immutable objects replication scheme are supported.

TABS/Camelot

TABS supports the nested transaction scheme. The operating system of each workstation provides a Transaction Manager which is responsible for coordinating the commit procedure. The fan-out approach is used, so a number of managers will have to cooperate in order to commit an action which spans multiple workstations. Each manager is responsible for handling the commit procedure on all affected objects which reside on its workstation, and acting as the coordinator for the managers of those workstations that are its children (with respect to the action). If an action aborts, the Transaction Managers are responsible for ensuring the partial effects of the action are undone. The undo/redo log is scanned and those entries corresponding to the action are examined and their effects 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 *non-blocking* commit which is a combination of the three-phase and the Byzantine commit protocols [Spector 87b]. The non-blocking commit procedure is relatively expensive compared to the blocking commit procedure; but, it also reduces the likelihood that an object will remain locked due to a failure during the commit procedure.

Both TABS and Camelot support a pessimistic synchronization scheme. A *type-specific locking* scheme [Korth 83, Schwartz 83] is used which enables a programmer to define his own lock modes and protocols so they can be tailored to the requirements of the objects. These locks can achieve greater concurrency within an object by allowing the semantic knowledge about its operations to be used. However, the serializability properties of actions cannot be enforced because the users can define their own synchronization rules. A time-out mechanism is used to detect deadlocks.

Neither a security scheme nor an object replication scheme are provided by either system.

An individual object cannot be recovered; instead, the entire workstation has to be recovered. A roll-back recovery scheme which uses an undo/redo log is supported. During the recovery procedure, the Recovery Manager of the failed workstation examines the log entries and queries the Transaction Manager to discover the state of the corresponding actions. The log records either the old values and new values of modified pages, or the operations which were invoked. If the log records old value/new value entries, each page of an object is reset to its most recently recorded value associated with an action which has committed. If the log records operation entries, the Recovery Manager determines whether the invocations need to be redone or undone, and issues the appropriate requests to the objects.

Chapter 5.
Object Interaction Management

Managing the invocations between objects is the second most important function of a DOBPS. In this chapter we describe the features provided by a system for handling object interactions, for locating server objects, and for detecting invocation failures. Finally, we present a brief overview of the manner in which object interactions are managed by a number of existing systems.

5.1. System-level Invocation Handling

When a client makes an invocation on an object, a DOBPS is responsible for performing the necessary steps to deliver the request to the specified server object and to return the results back to the client. A primary goal of DOBPSs is to perform inter-object invocations quickly and efficiently since they are usually a major source of overhead in these systems. The method by which invocations are handled at the system level depends entirely on the object model (see §3.2.) supported.

5.1.1. Message Passing

A DOBPS which supports the active object model typically also provides 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 and the message is sent to a server process or a port associated with the invoked object. A server process in the invoked object receives this 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 normally go through the effort and expense of setting up, using, and tearing down a connection. In addition, the binding of a client and a server is done dynamically on every invocation instead of just once in a separate initial phase. This is more suitable to the request/response communication pattern common to DOBPS. It also permits a system to more easily support features such as object scheduling and migration (see §6.2.).

A message passing scheme may either send messages directly to the specified objects or indirectly through some intermediate entity such as a port. The direct approach requires less performance overhead and eliminates the need for connection mechanisms between workstations. However, it is also less flexible than the intermediate entity approach. Intermediate entities act as an interface between a client and a server. In the port scheme, an object owns one or more ports to which clients send invocation request messages. Any client which has knowledge of a port can send a request to it; but, only a server process of the object which owns the port may remove a request from it. There are a number of advantages associated with using ports. First, a port can be an access point to a service, not to a particular server object. For example, multiple objects which provide the same service may be permitted to extract messages from the same port. Second, an object which owns multiple ports may use them to distinguish between its clients. For example, higher priority

clients can be given access to a special port, or different sets of operations can be made available from different ports. Third, a port may be owned by the network server object which can deliver message across inter-machine boundaries in a transparent fashion. For example, a request message sent to a port owned by a network server object can be transported to a port which resides on the same workstation as the invoked object. One problem with using ports is ensuring that only authorized server processes are able to extract requests from a port.

The message passing scheme is well suited for the multiple machine environment of a DOBPS because it maps naturally onto the mechanisms required for inter-machine communication. It also creates a clear boundary between all objects which does not permit an object to directly examine the internal representation of another. As a result, it satisfies one of the goals of object abstraction. A major disadvantage of the message passing scheme is that intra-machine invocations may be relatively expensive and slow compared to conventional procedure calls because of the extra, and unnecessary, overhead of creating messages.

x[12] When a DOBPS supports the passive object model, the pure message passing scheme should not be used (§3.2.2.). Furthermore, this scheme should not be used when the large, medium and fine-grain object model is supported (§3.1.).

√ The message passing scheme should be used when a DOBPS supports the active object model (§3.2.1.).

5.1.2. Direct Invocation

A DOBPS which supports the passive object model typically also provides the *direct invocation* scheme to handle object interaction. In the passive object model, a single process is responsible for performing all the operations to satisfy an action. As a result, a process will

[12] The definition of the symbols "x" and "√" can be found in §1.3. on page 6.

migrate from operation to operation and from object to object whenever it makes an invocation. In the direct invocation scheme, the system is responsible for locating an invoked object (see §5.2.) and then taking the appropriate steps to perform the invocation.

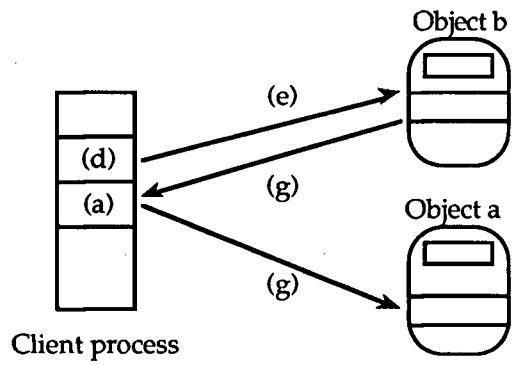

Figure 5.1: Direct invocation, local request

When a client process and the server object it invokes reside on the same workstation, the following four steps are taken. (See Figure 5.1).

(a) The state of the process and the object in which it currently resides are recorded in the stack of the process. The system may protect the stack to ensure that this information cannot be examined or corrupted by the activities of the subsequent invocation.

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

(e) The invoked object is loaded into memory and a subroutine call is made to start the process executing at the appropriate address in the object's code.

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

When a client process and the server object reside on different workstations, the following three steps must be added, the first two after step (a) and the third after step (e). (See Figure 5.2).

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

(c) The operating system which receives this message creates a worker process to execute on behalf of the original process.

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

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

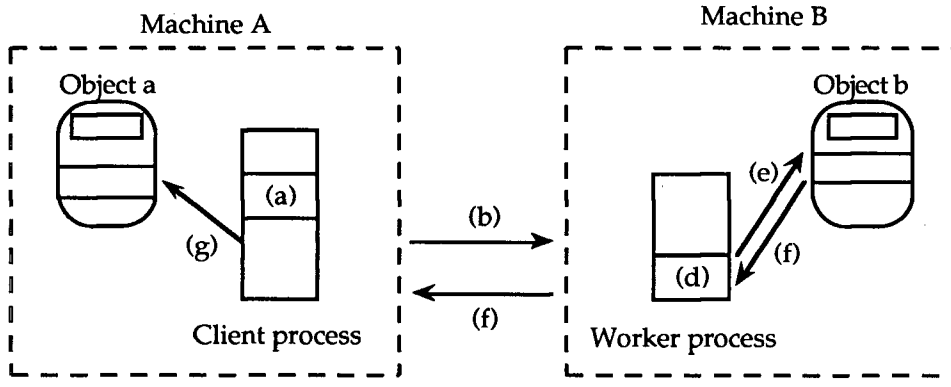

Figure 5.2: Direct invocation, remote request

An advantage of the direct invocation scheme is that it may incur less performance overhead than the message passing scheme. Interactions between objects which reside on the same workstation is relatively efficient compared to the message passing scheme. Remote invocations on the other hand, are identical to inter-machine communication in the message passing scheme. A disadvantage of this approach is that object migration is complicated. For example, there is the added complexity of moving an object that has been invoked by an action, but the process currently executes within some other subsequently invoked object. Therefore, when an object is moved, the

stack of a process which is executing in the object has to be updated accordingly to reflect the new location of the object.

- x When a DOBPS supports the active object model, the direct invocation scheme should not be used (§3.2.1.).
- √ A DOBPS which supports the passive object model should use the direct invocation scheme (§3.2.2.).

5.1.3. Passing Object Parameters

Information is transferred between interacting objects by passing parameters. Unique to DOBPSs are the parameters which reference objects. There are at least three schemes which can be used to pass parameters of this type: pass-by-reference, pass-by-value, and call-by-move.

The most common approach is the *pass-by-reference* scheme. In this scheme, the identifiers of the objects are simply passed as the parameters to a server object. The server object can then use the identifiers to make invocations directly on the objects in order to examine or modify them. The pass-by-reference scheme has the advantage that objects never have to be moved as is the case in the other two schemes. A drawback of this scheme is that the subsequent invocations which have to be made in order to obtain information can negate this savings, especially if the objects reside on different workstations or if a large number of invocations are made.

The *pass-by-value* scheme is another approach. In this scheme, a copy of each immutable object which is referenced in an invocation is moved to the workstation on which the server object resides. This allows the server object to make invocations on these object more efficiently since they will both reside on the same workstation. A disadvantage of the pass-by-value scheme is that it may be expensive if the objects which are transferred are large. In addition, the objects which are passed can only be examined, they cannot be modified because they are immutable.

A combination of the pass-by-reference and the pass-by-value schemes is the *call-by-move* scheme [Black 86b]. In this scheme, an object which is referenced in an invocation is moved to the workstation on which the server object resides. This allows the server object to examine and manipulate the object directly and efficiently. When the invocation completes, the object is returned to the workstation on which it originally resided. The savings, or expense, of the call-by-move scheme depends on the number of objects passed, the size of the objects moved, and the number of invocations made on the moved objects. The main problem with this scheme is that the DOBPS has to support an object migration scheme (see §6.2.2.) in order to move an object from one workstation to another. In contrast, the call-by-value scheme simply creates a copy of an immutable object on another workstation.

√ When a DOBPS supports the call-by-move parameter passing scheme, the object migration scheme (see §6.2.2.) must be used.

5.2. Locating an Object

A DOBPS provides the property of location transparency so that a client does not have be aware of an object's location in order to invoke the object. Whenever a client makes an invocation, the system is responsible for determining the workstation on which the object resides in order to deliver the request. Unfortunately, the task of maintaining accurate location information is complicated by two problems. The first problem is that a DOBPS may allow an object to move from one workstation to another (see §6.2.) at any time. The second problem is that object and workstation failures may occur (see §4.4.) which may make an object temporarily unavailable. There are a number of schemes which can be used to locate objects, some are described below.

The distributed environment of a DOBPS places a number of restrictions on the identifiers used to reference the objects. First, it is necessary for an object to keep a constant identifier

throughout its lifetime, otherwise references to it may be lost. Second, an identifier should never be reused, it should be unique in both time and space to ensure that confusion will not arise between two different objects. Third, for an object replication scheme (see §4.4.2.) to be transparent to the user, the system should provide all the replicas of an object with the same user-level identifier; however, each replica must have a unique system-level identifier. These restrictions influence the location scheme used by a DOBPS.

The straightforward scheme is to *encode* the location of an object *within its object identifier.* When an invocation is made, the operating 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 very simple and efficient scheme. Unfortunately, it has the deficiency that objects cannot change their locations once created and assigned to a workstation, since this would also require their identifiers to be changed.

- A DOBPS cannot support an object migration scheme (§6.2.2.) when the locations of the objects are encoded within their identifiers.

Another approach is the *distributed name server* scheme. A distributed name server is composed of a group of name server objects which are maintained on a number of different workstations. A name server object cooperates with its counterparts so that collectively they contain up-to-date information about the location of every object of the system. When an invocation is made, a location request is sent to one of the name server objects which determines (possibly with the assistance of other name server objects) the location of the specified object. An advantage of this scheme is that every operating system of a DOBPS does not have to maintain location information. Unfortunately, there are a number of disadvantages with using a distributed name server. A major problem is that the name server has to be notified every time a new object is created or an object moves from one workstation to another. Another problem is updating the server

is not instantaneous and maintaining consistent information amongst the multiple components of the name server is difficult. Finally, a network partition may separate a group of workstations from all name server objects. As a result, objects in these workstations will not be able to interact with one another. These problems often discourage the use of distributed name servers.

The *cache* location scheme is another alternative. In the pure cache scheme, each operating system maintains a cache which records the current location of every remote object referenced by an object which resides on the workstation. When a client makes an invocation and the invoked object does not reside locally, then the cache is examined to determine the workstation on which it resides. The major advantage of the cache scheme is that it permits the location of an object to be quickly and efficiently determined. In addition, a network partition does not prevent the objects of a group of isolated workstations from interacting with one other, as is the case in a distributed name server scheme. There are at least three problems with using a pure cache scheme. First, ensuring that each cache is constantly up-to-date is an expensive operation. For example, the operating system on each workstation in the network has to be notified whenever an object is moved from one workstation to another and their caches updated accordingly. Second, each operating system may potentially have to maintain a large amount of location information. Third, if the cache does not have an entry for an object, then its location is unknown and cannot be determined. As a result, the invocation has to fail.

x When a DOBPS supports an object migration scheme (§6.2.2), the cache location scheme should not be used due to the large overhead of updating the caches.

The *broadcast* location scheme is an approach which eliminates the overhead of maintaining a cache. When a client makes an invocation and the invoked object cannot be found locally, then a message is broadcast throughout the network requesting the current location of the object. Every operating system which receives this request searches its workstation for the object.

If it is found, a reply containing its location is returned. Otherwise, the request is simply ignored. The major advantage of using a broadcast scheme is its flexibility. An object can move from one workstation to another while avoiding the unnecessary expense and delay of having to notify the other workstations or a distributed name server. The major problem with this scheme is that it is relatively slow and inefficient. Broadcasts tend to clutter up the network, disturbing all workstations with location requests even though only one workstation is directly involved with each request.

The *cache/broadcast* location scheme is an amalgamation of the cache scheme and the broadcast scheme. The cache/broadcast scheme is more efficient than the cache scheme in that each workstation only maintains a small cache which contains the last known locations of a number of recently referenced remote objects. Furthermore, the totality and correctness of cache information is not guaranteed. This scheme is also more efficient than the broadcast scheme in that it reduces the number of broadcasts that need to be issued. Only if the location of a remote object is not found in the cache or the object does not reside at its last known location, will a location request be broadcast throughout the network. When an object's new location is determined, the cache of the workstation which made the location request is updated. The cache/broadcast scheme has some of the efficiency of the cache scheme and all of the flexibility of the broadcast scheme. The drawback is that the mechanisms of both schemes must be provided.

Forward location pointers can be used to enhance almost any location scheme. A forward location pointer indicates the new location of an object which has moved. Whenever an object is moved from one workstation to another, a forward location pointer is left at the first workstation. To locate an object which has moved, the forward pointer, or chain of pointers, is followed to the workstation on which the object currently resides. The main problem with using forward location

pointers is that they may be garbage collected or lost due to workstation failures. They also introduce additional overhead and the garbage collection problem.

5.3. Detecting Invocation Failures

An invocation failure may be classified as being either an *existing fault* or a *transient fault* [Ahamad 87a]. Existing faults are failures which occur before an invocation is made, thereby resulting in a server object which is not locatable. These faults are relatively easy to detect and handle. If an object cannot be located, the invocation simply fails. Transient faults on the other hand, are failures which occur while an invocation is being performed. They are failures which occur sometime after a server object has accepted an invocation request, but before the modifications made to it have been made permanent. These faults are much more difficult to detect and handle because of the many different ways an invocation can fail. For example, the object or the workstation of the client may fail, or the server may fail, or a network partition may occur. A DOBPS should provide mechanisms for both the client and the server to detect and recover from transient failures.

5.3.1. Client

The failure of an invocation has to be detected by the client, otherwise the corresponding action may block and wait indefinitely. When a client detects an invocation failure, the recovery procedure is initiated. The recovery procedure may release the resources held by the client and notify the corresponding action of the failure, or it may reissue the invocation request after some period of time. For example, the invocation request can be reissued if a replication scheme (see §4.4.2.) is supported. There are a number of schemes which can be used by a client to detect the failure of an invocation, some are described below.

The straightforward approach is for a client to assume its invocation has failed after some period of time has elapsed. This is referred to as the *time-out* scheme. The difficulty with the time-out scheme is determining how long to wait before concluding a failure has occurred. If it is too short, it may lead to unnecessary failures. If it is too long, then the client may wait unnecessarily, possibly tying up valuable resources while it is waiting. Unfortunately, there are many reasons why a client does not receive a result after some period of time, some are given below.

(a) The server object did not receive the request because of an object, a workstation, or a network failure.

(b) The server is still processing the request.

(c) The server was processing the request when it failed.

(d) The server returned a result but because of a network partition it has not been received.

The major problem with the time-out scheme is that the client does not determine whether or not the invocation has actually failed, it simply assumes the invocation has failed.

x When the clients use a time-out scheme, the servers cannot use a probe scheme (§5.3.2.).

An extension to the time-out scheme is the *probe* scheme, in which a client periodically checks the server object to determine if it has failed. There are two variations of the probe scheme: the object probe scheme, and the invocation probe scheme. The simpler of the two scheme is the *object probe* scheme. In this scheme, after a time-out period has expired, the client issues a probe request to the server object inquiring if it is still operational. An object which receives a probe request immediately returns an acknowledgement. If the client receives an acknowledgement, it waits for another time-out period. However, if there is no response to a number of probe requests, the client concludes that the server and its invocation have failed. The object probe scheme has the same problem as the time-out scheme in that it does not determine if the invocation has actually failed, it simply checks the status of the server object. This may lead to a client waiting

indefinitely for a result that will never arrive. For example, the server object may actually have failed, but was recovered using a roll-back scheme (see §4.4.1.1.) before the failure was detected by the client. Similarly, a network partition which is detected by the server but is fixed before the client detects it, may result in the server destroying the orphaned invocation (see §5.3.2.) and the client waiting indefinitely.

In the *invocation probe* scheme, a client contacts the server object to inquire about the status of a particular invocation. After a time-out period has expired, the client issues a probe request to the server object. An object which receives a probe request immediately determines if a process is executing on the inquirer's behalf. If there is, an acknowledgement is returned and the client waits for another time-out period. Otherwise, a negative acknowledgement is returned and the client is notified of the failure. If a client does not receive a response to a number of probe requests, it concludes that the server and its invocation have failed. The advantage of the invocation probe scheme is it better determines whether or not an invocation has actually failed. A problem with this scheme is its relatively large overhead. For example, every time an object receives a probe it must suspend its activities in order to process the request. In addition, each object has to maintain a record of the clients that are associated with its processes.

- x When the clients use either of the probe schemes, the servers should not use the time-out scheme (§5.3.2.).
- √ The servers should use the probe scheme (§5.3.2.) when the client uses either of the probe schemes.

5.3.2. Server

The failure of an invocation should be detected by the server object, otherwise valuable resources may be tied up indefinitely. An invocation which has started but its results are no longer

wanted is referred to as an *orphan*. An orphan may arise due to the failure of a client, an aborted transaction, an object failure, a workstation failure, or a network partition. Orphans waste system resources, such as holding locks or causing a loss of throughput, therefore they should be eliminated as quickly as possible. For example, an action may be delayed because a required lock is held by an orphan. The recovery procedure should release the resources held by an orphan, undo the modifications it made to the object, and destroy the server process which may possibly still be performing the invocation.

A *time-out* scheme, similar to the one used by the clients, can be used by the server objects to detect invocation failures. When an action first modifies an object, a clock at the workstation on which the object resides is initiated. After the time-out period has expired, the invocation is concluded to be an orphan. Unfortunately, this scheme has the same problem as the corresponding scheme used by the clients, namely it may cause an action to abort when it is not necessary.

x The servers should not use the time-out scheme when the clients use a probe scheme (§5.3.1.) because all the mechanisms exist for the server to use the probe scheme.

When the clients use a probe scheme, the server objects can use the *probe* scheme to detect failures. A server object which does not receive a probe request from a client after a long period of time concludes that the client has failed and the corresponding invocation is an orphan. The advantage of this scheme is that it better determines when an invocation has failed.

x The servers should not use the probe scheme when the clients use a time-out scheme (§5.3.1.).

√ The servers should use the probe scheme when the clients use either of the probe schemes (§5.3.1.).

An alternate scheme is to periodically pass *status reports* among the workstations of the network. The information in the report can contain a list of all workstations or all actions which

are known to have failed. When a workstation receives a report, it checks its processes against the list to determine if any of them have clients which were executing on a workstation which has failed, or if any of them are part of a failed action. These processes are orphans. The status report scheme may have a very high overhead because of the potentially large amounts of information which may have to be maintained and passed among the workstations. In addition, migrating an object is made much more complex. When an object is moved, every server object that the object invoked has to be notified of its new location.

- x When a DOBPS supports the object migration scheme (§6.2.2.), the status report scheme should not be used because of the added complexity of moving objects.

5.4. An Overview of Object Interaction Management in Existing Systems

Amoeba

Amoeba supports the message passing scheme which uses ports. Server authorization is done using special hardware (see §4.5.) which ensures that only valid objects can open and hence extract requests from a port.

The cache/broadcast scheme is used to locate the ports of an object. When information in the cache is found to be incorrect, a location request message is broadcast over the network. If this message reaches the specified object, the object responds. The cache is then updated and the request is sent.

The object probe scheme is used by both the client and the server 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 after some period of time after the acknowledgement, it sends an "are you alive?" message to the server object which immediately

responds if it is able. If the client does not receive a reply to this message, it concludes that the server and the invocation have failed. Similarly, if the server stops receiving "are you alive?" messages from a client, it concludes that the client has failed and kills the corresponding orphan invocation.

Argus

Argus supports the direct communication variation of the message passing scheme. To aid 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 which made the invocation.

The locations of the objects are encoded within their identifiers.

The invocation probe scheme is used by the clients 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 which it invoked. An object which 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 acknowledgement that the invocation is still being executed is returned. Otherwise, a negative acknowledgement is returned. If the client receives a negative acknowledgement or it fails to receive an acknowledgement, then it concludes that the invocation has failed.

The status report scheme is used by the servers to detect invocation failures. Each large-grain object maintains a number of data structures in secondary storage 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 and is incremented every time the object recovers from a failure. The done list records all actions which are known by this object to have failed. The map list records the latest known crash counts of all large-grain objects known by this 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 which 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 workstations have failed recently. If any have, the object determines whether any of its current clients resided on one of the failed workstations. All orphans are then destroyed. The done and the map lists of the large-grain object which received this information are then updated to reflect the changes in the system. A detailed description of this status report scheme used is given in [Walker 84].

CHORUS

CHORUS supports the message passing scheme which uses ports. A port is either open or closed. Requests may be sent to or extracted from an open port, but not a closed one. Each port may be dynamically associated with different objects, though only one object can be associated with an open port at any one time. An object which opens a port is its current owner and is the only one which may extract messages from it. A port which is no longer required may be close, allowing it to be subsequently opened by another object. Ports are bi-directional in that messages sent by and received by an object pass through one of its ports. Inter-machine communication is done by passing messages to a surrogate local port which is managed by a network server. The network server handles 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 in each port identifier is the identity of the workstation on which the port was created. In addition, each operating system maintains 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 which maintains the information. A location request is then sent to the operating system of that workstation. If this information cannot

be obtained, because the machine could not be contacted or the information is found to be incorrect, a location request is broadcast throughout the network.

The time-out scheme is used by the clients, while no scheme is used by the servers to detect invocation failures.

Clouds

Clouds supports the direct invocation scheme. When an invocation is made on a local object, the states of the process and the object are recorded on the stack of the process. This stack is made inaccessible, a new stack is created for the process, and the parameters of the invocation are added the new stack. The invoked object is then loaded into memory and a branch to the appropriate entry point of the object is performed. When the operation terminates, the result parameters are passed to the operating system which returns them to the client process. The previous stack is then used to restore the process to its state prior to the invocation request.

When an invocation is made on an object which does not reside locally, the communication manager of the workstation 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 which receive a "search and invoke" request determines if the object resides on its workstation. If the object cannot be 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, the worker process passes the results to its communication manager which

constructs a reply message. This message is then sent back to the communication manager of the first workstation, the client process is unblocked, and the results are passed to it.

A broadcast scheme is used to locate objects. The workstation on which the client process resides is first searched to determine if the invoked object resides locally. If the object reside on a remote workstation, a "search-and-invoke" message is broadcast to the other workstations of the network. The search procedure is made more efficient by the use of three structures: an active object table, a maybe table, and an object directory. The active object table records all operational objects that reside on the workstation. The maybe table indicates that either the object does not reside locally or that it might be. The object directory records all operational and dormant objects that reside on the workstation. The search procedure examines each structure in turn when it attempts to locate an object. The rational for performing multiple tests is so that a number of quick checks can be made before resorting to the task of examining the entire workstation. Details of this broadcast location scheme is given in [Pitts 87].

The object probe scheme is used by the clients to detect invocation failures. Periodically, each operating system probes its counterparts the network to determine if they are still functioning. An operating system which does not respond after a substantial period of time is concluded to have failed. All uncommitted actions which sent an invocation request to a workstation which has failed, are aborted.

A variation of the time-out scheme is used by the servers to detect invocation failures and orphans. 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

released. To alleviate the problem of an action aborting unnecessarily, a refresh protocol is used to periodically advance the quiesce and release times of each action which is still functioning. Details of this time-out scheme is given in [McKendry 85].

Eden

Eden supports the message passing scheme. The object which sends the message is responsible for packing and encoding the parameters into their self-describing, external data structures. The object which receives the message is responsible for decoding and unpacking the parameters. Using an external data structure enables machines with different data representations to be used; but, it also restricts the parameters to a fixed set of system defined types.

A pure cache scheme is used for locating remote objects.

The time-out scheme is used by the clients, while no scheme is used by the servers to detect invocation failures.

Emerald

Emerald supports the direct invocation scheme which uses a call-by-move parameter passing scheme. Whenever a process makes an invocation on an object, a new activation record is created for the process. If the invocation is made on an object which resides locally, the activation record is appended to the top of the process' current stack. If it is made on an object which resides on a remote workstation, the activation record forms the base of the new process' stack on that workstation. Therefore, a single process may have a stack that is distributed across several workstations.

To locate objects, a cache/broadcast scheme which uses forward pointers is supported. Each operating system maintains a cache which records the last known location of remote objects. The location of an object is coupled with a timestamp to indicate the relative age of the data. When an

invocation is made, the cache is examined. If the cache information is found to be incorrect but the second operating system knows of a newer location (by checking values of the timestamps), then that location is checked. If the location of the object cannot be determined, a broadcast location request is issued. Emerald ensures that every operating system of the network receives this request and that all available workstations are searched.

The time-out scheme is used by the clients, while no scheme is used by the servers to detect invocation failures.

TABS/Camelot

Both TABS and Camelot support the message passing scheme which uses ports. Transparent inter-machine communication between a client and a server is provided by a pair of communication managers. The communication managers supply two ports: one of which resides on the workstation of the client and the other which resides on the workstation of the server. 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.

The broadcast scheme is used to locate objects. Each operating system maintains a list of all objects which reside on its workstation. When an invocation is made and the object does not reside locally, the operating system broadcasts a location request to its counterparts in the network. The operating system of the workstation on which the object resides returns an acknowledgement.

The time-out scheme is used by the clients, while no scheme is used by the servers to detect invocation failures.

Chapter 6.
Resource Management

A DOBPS, like any other distributed operating system, has to provide mechanisms to manage the physical resources of the network, including: primary memory, secondary storage devices, processors, and workstations of the network. In this chapter we are concerned with resource management as related to objects. We describe the representation of objects in memory and in secondary storage, and the methods by which they are transferred between these two resources. We then describe the features provided by a DOBPS for assigning objects to processors and for moving objects from one processor to another. Finally, we present a brief overview of the manner in which resource management is handled in a number of existing systems.

6.1. Memory & Secondary Storage

Two very important physical resources of a DOBPS which must be managed are the memory of the workstations and the secondary storage devices of the network. A persistent object which is maintained both in memory and in a secondary storage device is said to be *operational*. A

persistent object which is maintained solely in a secondary storage device is said to be *dormant*. An invocation made on a dormant object causes a volatile version or copy of the object to be created and loaded into memory to make the object operational. If necessary, new processes are created for the volatile version. When an operational object remains idle for a period of time, it can be made dormant in order for the system to reclaim the memory it occupies. An object is made dormant by copying its volatile version onto secondary storage and then deleting the volatile version, or simply deleting the volatile version if the modifications made to it have already been recorded. This automatic loading and unloading of objects is performed in a transparent fashion by a DOBPS in order to give the appearance that objects are always operational and ready to be invoked.

The secondary storage versions of persistent objects cannot be modified directly. Instead, when a persistent object is modified, the changes are first made on a volatile version of the object. These changes are not made on the object's version in secondary storage until the corresponding action commits. This approach guarantees that a consistent state of each object is always recorded. Consequently, if an object or the workstation on which it resides fails, only the volatile version of the object is lost and the object can be restored to the state of the secondary storage version. Another advantage of this approach is that when an action aborts, the volatile versions of the objects it affected can simply be discarded to undo its modifications.

To enhance its memory and secondary storage mechanisms, a DOBPS can supply paging hardware. An object will then be composed of a number of pages, organized by a page table which links the pages of the object. This enables a system to support a demand paging scheme so that only those pages which are required are loaded into memory. Furthermore, only those pages which are modified need to be written to secondary storage. A paging scheme can be incorporated into almost any memory or storage scheme and can substantially reduce the overhead of maintaining objects, especially if the objects are large.

6.1.1. Representation of Objects in Memory

An object's representation in memory depends both on the synchronization scheme (see §4.2.) and the action management scheme (see §4.1.) supported by the DOBPS. The synchronization scheme influences the number of versions of each object that are maintained, while 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. The first action that makes an invocation on a dormant object causes the system to create and load a volatile version of the object into memory. That action and all subsequent actions which make invocations on the object modify the same volatile version. Maintaining only a single version of each object simplifies the synchronization mechanism since all the required control structures will be at one location.

x[13] When a DOBPS supports an optimistic synchronization scheme (§4.2.), objects should not be represented in memory using the single version approach.

√ The single version approach should be used when a pessimistic synchronization scheme (§4.2.) is supported.

When an optimistic synchronization scheme is supported, typically *multiple versions* of each object are created and maintained in memory. Every action which makes an invocation on an object causes the system to create a volatile version of the most recent secondary storage version of the object. Therefore, each action is assigned its own volatile version of an object on which to perform its modifications. This enables multiple actions to invoke the same object simultaneously and ensures they will not interfere with each other.

x When a DOBPS supports a pessimistic synchronization scheme (§4.2.), objects should not be represented in memory using the multiple version approach.

---

13 The definition of the symbols "x" and "√" can be found in §1.3. on page 6.

√ The multiple version approach should be used when an optimistic synchronization scheme (§4.2.) is supported.

The representation of each version depends on whether or not the nested transaction scheme (see §4.1.) is supported by the DOBPS. The traditional approach of representing a version in memory as an exact copy of the corresponding object is sufficient in DOBPSs which support a request or a transaction scheme. In these schemes, if an action fails, the version can simply be discarded. This approach, however, is usually not adequate when a nested transaction scheme is supported. In the nested transaction scheme, multiple sub-actions may modify the same version, with each completing or failing independently. If a sub-action partially modifies a version and is then forced to abort, the changes it made have to be undone so that the version is restored to the state before the sub-action started. In particular, the modifications made by sub-actions which successfully completed before the failure occurred must be recovered. To undo the changes made by a failed sub-action, an additional mechanism, such as one which maintains an undo/redo log (see §6.1.2.2.) or an immutable *hot standby* that records the state of the version created by the last sub-action to successfully complete, is required.

In the *undo/redo* approach, a version is represented in memory as an exact copy of the corresponding object. Modifications made by sub-actions are all performed directly on the version with the changes recorded in an undo/redo log. If a sub-action fails, the entries in the log are used to undo the effects of the sub-action on the version. If the failure causes the version to be lost or to be corrupted beyond repair, a copy of the version maintained on secondary storage is obtained and the effects of all the previously completed sub-actions are redone. The overhead of maintaining an undo/redo log and the relatively high expense of the recovery procedure are two problems of this approach.

One variation of the hot standby approach is the *version manager* scheme. In this scheme, a special object is created for each version which is affected by a nested transaction in order to manage and control access to the version. Sub-actions reference a version via its manager object. When a sub-action first modifies a version, the corresponding manager object creates a temporary copy of the hot standby. Modifications made by the sub-action are made on its copy. If a sub-action successfully completes, the manager objects makes its temporary copy the new hot standby. The overheads of creating manager objects and managing multiple copies of each version are two disadvantages with the version manager scheme. The advantage of this scheme is that it may be combined with an optimistic synchronization scheme.

Another variation of the hot standby approach is the *stack of versions* scheme [Liskov 83b]. Each object is represented as a stack consisting of intermediate versions with the most recent version of an object on the top of its stack. When a sub-action first modifies an object, a copy of the object's most recent version is made and appended to the top of its stack. Modifications made by this sub-action are then performed on this version. If the sub-action successfully completes, the stack is left for the next sub-action. If the sub-action fails, the state at the top of the stack is removed to restore the object to the state before the failed sub-action. The drawback of the stack of versions scheme is that only a single sub-action can modify an object at a time. Consequently, a pessimistic synchronization scheme should be supported when this scheme is used.

6.1.2. Representation of Objects in Secondary Storage

The primary reason for maintaining a copy of each persistent object in secondary storage is to enable a DOBPS to recover from object and workstation failures (see §4.4.). A system must record enough information so that a persistent object can be restored to a consistent state should it fail. A DOBPS may record the entire state of a modified object or it may record the changes made to each

object since some previously recorded state. Both of these schemes have their benefits and drawbacks which will be discussed below.

6.1.2.1. Checkpoint Schemes

In the *checkpoint* scheme, the entire state of each object modified by an action is recorded onto secondary storage when the action commits. In the pre-commit stage of the commit procedure, the modified version of an object is written to secondary storage, without disturbing the old version. In the commit stage, the modified version replaces the old version in one atomic step and the old version is discarded. If the action aborts, the modified version is simply discarded and the old version remains unaffected. The pure checkpoint scheme makes efficient use of secondary storage since only a single copy of each object is maintained. Unfortunately, this also has the drawback that it does not allow a previous checkpoint of an object to be examined. Another problem with the checkpoint scheme is that it may have a high performance overhead since the entire state of an object has to be recorded whenever the object is modified. This is especially true if only a small change is made to an object with a very large state. Some of this performance overhead, however, can be reduced if a paging scheme is also supported by the DOBPS.

- x The checkpoint scheme does not provide the features required by the optimistic synchronization scheme (§4.2.) or the roll-forward recovery scheme (§4.4.1.).
- √ When the checkpoint scheme is supported, the roll-back recovery scheme (§4.4.1.) has to be used.

A minor 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 and it is then marked as the most recent

version. The history of checkpoints scheme enables the previous checkpoints of an object to be examined. This is a feature required by DOBPSs which support an optimistic synchronization scheme (see §4.2.). There are at least two disadvantages of this scheme: additional storage space must be used in order to record the extra checkpoints, and the garbage collection problem is introduced.

x The history of checkpoints scheme does not provide the features required by the roll-forward recovery scheme (§4.4.1.).

√ When the optimistic synchronization scheme (§4.2.) is supported, the history of checkpoints scheme should be used. In addition, when the history of checkpoints scheme is supported, the roll-back recovery scheme (§4.4.1.) has to be used.

6.1.2.2. Log Schemes

In a *log* scheme, the changes made to persistent objects are recorded in a common log which is maintained in secondary storage. Whenever an object is modified, usually one or more entries are added to the log to record the changes made. The amount of information which has to be written to secondary storage depends on the particular logging scheme supported. However, the expense and overhead of recording these entries is typically less than recording a checkpoint of the object. A problem with the log scheme is that a large log increases the overhead of object maintenance and recovery (see §4.4.1.). Consequently, the log should be cleared periodically.

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 status of the actions which made these changes. When an object is modified, the relative or absolute changes made to the object are recorded in the log, along with an identifier indicating the action which made the modification. Enough information is maintained so that all the modifications made on

an object by an action which committed can be redone by the recovery procedure. For example, a log can record the new values of the updated data or the operations that were performed on the objects. Periodically, a new base checkpoint for each object is recorded and its entries are cleared from the log. This is a very simple and efficient scheme.

A variation of the redo log is the *undo/redo* log. Enough information is maintained for the effects of an aborted action to be undone and for the effects of a committed action to be redone. For example, a log can record both the old value and the new value of an updated object. For the recovery procedure to execute successfully, a write-ahead logging protocol [Gray 78] should be used. The write-ahead protocol guarantees that the log entries are safely recorded in the log before an action commits and before a checkpoint is made. The undo/redo log differs from the redo log in that a workstation-wide checkpoint of all operational objects is made. To take a checkpoint, all operational objects which reside in the workstation are suspended when the operations they are performing terminate. The objects are then checkpointed and the activities resumed. An advantage of this scheme is that it ensures that a consistent global state is recorded. The primary drawback of this logging scheme is that all activities usually have to be suspended until the checkpoint is taken.

√ When an undo/redo log is used, the roll-back recovery scheme (§4.4.1.) should be supported.

A different type of log is the *commit log* [Oki 85]. A commit log records the states of all objects which were modified by actions that are in the process of being, or have been, committed. It also records the stage of the commit procedure reached by the actions. An entry in the log can be either a data entry or an outcome entry. A data entry records the entire state, or checkpoint, of an object. An outcome entry records the last known stage of the commit procedure performed by an action: pre-commit, commit, or abort. In the pre-commit stage of the commit procedure, a

checkpoint of each object modified by the action being committed is written to the log, each in its own data entry. This is followed by pre-commit 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. Enough information is maintained so that the commit procedure can be restarted if a failure occurs. Periodically the log is cleared of all entries which are outdated.

6.1.3. Destroying Objects

When an object is no longer required, it can either be explicitly deleted by its owner or garbage collected [Dijkstra 78] by the system. When the *explicit deallocation* scheme is supported, the users are responsible for periodically destroying unwanted objects. A benefit of this scheme is a DOBPS does not have to supply an independent file server since this feature is now provided automatically. Unfortunately, determining when an object is no longer required may be difficult due to the multiple user environment of a DOBPS.

When the *garbage collection* scheme is supported, the task of destroying unwanted or unreferenced objects is performed automatically by a DOBPS. Unreferenced objects are defined as those objects whose identifiers are not owned by any other object. Most garbage collection mechanisms use a variation of the following mark-and-sweep algorithm to find unreferenced objects.

(a) Initially, all objects are marked *white*.

(b) Objects that are known to be referenced (ones containing executing processes, for example) are then marked *gray*.

(c) Each gray object is scanned and all objects referenced by it are marked gray. The scanned object is then marked *black*.

(d) When all gray objects have been scanned, the system will consist of black objects (those objects which are referenced) and white objects (unreferenced objects) which can be deleted.

Unfortunately, garbage collection in a DOBPS is a very difficult and expensive task due to the multiple machine environment. There are two types of garbage collection schemes: local collectors, and global collectors. A local garbage collector is responsible for periodically searching a workstation for objects which were, but are no longer, referenced by other local objects and therefore can be destroyed. A global garbage collector is responsible for periodically searching all the workstations of the network to determine the objects which are no longer referenced. A problem with global garbage collection is that the collector may encounter hardware failures which make the marking procedure difficult. For example, a network partition can at least temporarily make communication between some of the workstations impossible.

The explicit deallocation scheme is typically used in DOBPSs which support large-grain objects because a user can better and more efficiently determine when a large-grain object is no longer required. Conversely, a local garbage collection scheme is typically used in DOBPSs which support medium-grain and fine-grain objects.

6.2. 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 facts: first, objects should be assigned to different, lightly-loaded processors so that they can execute concurrently; second, objects which 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 costs for inter-machine communication. Consequently, the objects of an object-based program should probably be assigned to a group of closely-spaced, lightly-loaded processors for optimal performance.

Scheduling the use of the processors can be decomposed into two distinct tasks: object scheduling to initially assign objects to processors, and object migration to move objects from one processor to another once their characteristics are known. A DOBPS must supply an object scheduling mechanism, but it may not supply an object migration mechanism. In addition, it may permit either the user or the system to be responsible for determining where to assign objects, although this may depend on the user's view of the distributed environment (see §6.3.).

6.2.1. Object Scheduling

When a new object is created or a dormant object is made operational, it has to be assigned to a processor. A newly created object can usually be assigned to any processor of the system, unless for performance reasons it must reside on the same workstation where certain devices are attached. An object which is made operational, however, may have some restrictions placed on where it is assigned. In particular, once an *immobile* object has been assigned to a processor it cannot change its location. In this case, an object is always re-assigned to the same processor on which it was originally created. A more general restriction may be that an object can only be re-assigned to a processor that is of the same type as the one on which it was originally created. There are two variations of the object scheduling scheme: explicit object scheduling, and implicit object scheduling.

The *explicit* object scheduling scheme permits a user to explicitly specify where to assign an object. If no processor is specified, the object may simply be assigned to the user's processor. Both

the primary advantage and disadvantage of explicit object scheduling is that a user has to be aware of the distributed computing environment. This scheme permits a user who is knowledgeable about the characteristics of his objects and the workstations to determine the best locations to place his objects in order to improve the performance of his programs. For example, a user can assign his objects to lightly-loaded processors. Unfortunately, a user who is ignorant of the distributed environment may end up with a single processor containing all of his objects. This eliminates the major purpose of providing a distributed computing system.

- x When the explicit object scheduling scheme is supported, the distributed environment cannot be totally hidden from the user.
- √ The distributed environment must be either partially visible or totally visible to the user (§6.3.) when the explicit object scheduling scheme is supported.

In the *implicit* object scheduling scheme, the DOBPS is entirely responsible for determining where to assign the objects. This decision can be made either by a single processor, *autocracy*, or by all the processors, *democracy*. Usually, the more centralized the scheduler, the simpler the scheduling algorithm and the smaller its communication costs will be. On the other hand, the more distributed the scheduler, the more reliable it will be. Most schedulers are neither entirely autocratic nor democratic, but instead take a more intermediate approach to balance the added costs with the benefits gained.

A great deal of research has been done to develop implicit processor schedulers. Unfortunately, most of it is theoretical dealing either with graph theory [Stone 77, Stone 78] or queueing theory [Chow 79, Agrawala 82]. In addition, these algorithms often make unrealistic assumptions such as knowing in advance the processing time required by each process or the amount of inter-process communication which will be performed. Therefore, much of this work is of little

or no use. Fortunately, there are a few practical implicit object scheduling algorithms, including: wave scheduling [Wittie 80, Van Tilborg 81], contract bidding [Smith 79], and tokens [Tripathi 86].

An implicit object scheduling mechanism may be very simple or very complex. A simple scheme will assign an object to the user's processor or to the processor nearest to the secondary storage device on which the dormant object resides. A more complicated scheme will examine the loads of the processors to determine the best location to place an object. In this case, when a single object is created or made operational, the object is typically assigned to the processor with the lightest load. When multiple objects such as an object-based program are created, a cluster of lightly-loaded processors are found and then each object is assigned to one of the processors. A problem with using load information is the cost of obtaining and maintaining accurate information can be relatively expensive. For example, the loads of the processors have to be estimated frequently. If accurate information is not maintained, there is the problem that a processor which is observed to be lightly-loaded may have a number of objects assigned to it, subsequently overloading the processor.

6.2.2. Object Migration

An *object migration* scheme[14] permits operational objects to move or migrate from one processor to another at any time, even while they are executing invocations. The work performed by an object which is moved is not lost nor is the action corresponding to it aborted. There are at least two advantages of object migration: 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 shutdown for maintenance to one which is working. It also enables

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

objects which interact heavily to be moved to the same workstation so that future communication costs can be reduced.

There are a number of existing process migration algorithms, including: adaptive bidding [Stankovic 84] and pairing [Bryant 81]. Unfortunately, migrating an executing process from one processor to another in a conventional distributed system is a difficult task. While it is not difficult to move the code which is being executed, the classical problem is moving the machine-dependent information such as the values of running clocks, the logical communication paths, and the data structures which are maintained in memory. Fortunately, many of these mobility problems are simplified by a DOBPS. Objects clearly define the entities which can be moved and encapsulates the components which should be moved as a unit. In addition, machine-dependent information is kept to a minimum, and the property of location transparency permits a DOBPS to automatically determine the new locations of objects which are moved.

The object migration scheme has a number of problems, nevertheless. First, the cost of moving an object may outweigh the benefits of the move. For example, moving an executing object from one workstation to another can be an expensive task, and an object cannot do any processing while it is being moved. Second, messages sent to an object while it is being moved should be accepted by one of the operating systems involved in the move and forward to the object when it becomes operational again. Third, an object should not be continuously moved about the network so that it gets little processing done. Finally, replicas of an object (see §4.4.2.) should not be moved to the same processor.

Most object migration mechanisms attempt to reduce the loads of heavily-loaded processors. Typically, 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 operational objects. Nearby processors are usually searched before ones that are a greater distance away are searched. This minimizes the

distance that an object is moved so that it will remain relatively close to the objects it interacts with. If an appropriate processor is found, the system determines which objects are to be moved, if any. This decision may be based on the size of the object, its estimated remaining processing time, the number of times it has already been moved, or the cost of moving the object. A number of operational objects are then moved from the heavily-loaded processor to the less loaded processor.

- When the object migration scheme is supported, the locations of the objects cannot be encoded within their identifiers (§5.2.). Furthermore, a DOBPS supporting object migration should not use the cache scheme to locate objects (§5.2.) or the status report scheme to detect invocation failures (§5.3.2.) because they rely on machine-dependent information.

6.3. Workstations

The final resource a DOBPS must manage are the workstations. The way a DOBPS manages its workstations limits the users' view of the distributed environment. Conversely, the extent to which a user is permitted to view the distributed environment influences the features that are provided by the system.

At one end of the spectrum, the distribution is *totally hidden*. A user views the system as a large, powerful, centralized processing entity and is not aware of the multiple-machine environment. In this approach, a DOBPS is entirely responsible for handling all low-level details dealing with the distributed environment. For example, the system has to manage an object scheduling scheme (see §6.2.1.) and an object replication scheme (see §4.4.2.). The main drawback of this approach is a user has no control over the system. For example, a user cannot specify to which processors his objects should be assigned.

x When the hidden distribution scheme is supported, the explicit object scheduling scheme cannot be used (§6.2.1.).

√ The implicit object scheduling scheme (§6.2.1.) should be used when the hidden distribution scheme is supported.

In the intermediate approach, the distribution is *partially visible.* A user may be aware of the multiple-machine environment; but, he will not be aware of the details of the individual workstations. For example, a user may be permitted to request that his objects be created on different processors but cannot specify the particular processors. When the distribution is visible, the user can be responsible for managing an object scheduling scheme (see §6.2.1.) or an object replication scheme (see §4.4.2.). However, this does not exclude the system from supplying these features.

At the other end of the spectrum, the distribution is *totally visible.* A user is fully aware of the underlying details of the system and he may have full control over the system. This knowledge may be used to optimize the performance of applications. For example, a user may be permitted to specify the particular processors on which his objects are to be assigned. In the totally visible approach, the DOBPS usually retains responsibility for handling the low-level details dealing with the distributed environment. However, the users may be responsible for providing some of the features related to distribution. The exact amount of responsibility the user has depends on the particular DOBPS used. The main advantage of this approach is additional flexibility. It permits users who want to ignore the distribution to do so while permitting others to exploit the distribution.

x When the totally visible distribution scheme is supported, the implicit object scheduling scheme should not be used (§6.2.1.).

√ The explicit object scheduling scheme (§6.2.1.) should be used when the totally visible distribution scheme is supported.

6.4. An Overview of Resource Management in Existing Systems

Amoeba

Amoeba represents each object in memory using the multiple versions approach and in secondary storage using the history of checkpoints scheme. Paging hardware is supplied to enable objects to be composed of a number of pages which can be mapped in and out of the object's address space. When an action examines or modifies an object, the demand paging scheme creates a page-for-page copy of its most recent version in memory. When an action attempts to commit, the old checkpoints of the object are examined to determine if there is a serialization conflict (see §4.5.). 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 supplies an explicit deallocation scheme.

The explicit object scheduling scheme is supported. A system maintains a collection of shared processors called a processor pool from which a user may dynamically request to use a number of idle processors; however, he can not specify which particular ones. When a processor is no longer required, it is returned to the pool for others to use.

A partially visible distributed environment is provided.

Argus

Argus represents each object in memory as a single version using the stack of versions scheme 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 operational objects. This checkpoint scheme differs from most workstation-wide checkpoint schemes in that it does not suspend all activities in the workstation until the checkpoint procedure is complete. Instead, the objects are permitted to perform invocations while other objects are checkpointed. The following steps are taken to perform a checkpoint.

(a) The objects of the workstation are notified that the checkpoint procedure has started.

(b) When all the operations of an object terminate, the object suspends itself and records a checkpoint in the new commit log. The activities of the object are then resumed. If an action performs the commit procedure, the corresponding log entries are recorded in the old commit log.

(c) When all the operational objects are recorded, the outcome and data entries that were recorded in the old commit log while the checkpoint procedure was executing are copied to the new commit log.

(d) The old commit log is then discarded.

Both the explicit deallocation and the garbage collection schemes are supplied. A local garbage collector is used to reclaim unreferenced medium-grain objects, while large-grain objects are explicitly deleted.

An explicit object scheduling scheme is supported. Objects are immobile, however, since their locations are encoded within their identifiers.

A totally visible distributed environment is provided.

CHORUS

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

An explicit object scheduling scheme is supplied. An object can close all of its ports and allow them to be opened by another object which may reside on another workstation. All subsequent messages to these ports can be then be examined by the new object.

The distributed environment is totally visible to the users.

Clouds

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

Objects exist until they are explicitly deleted.

An implicit object scheduling scheme is supported and total hidden distributed environment is provided.

Eden

Eden represents each objects in memory using the single version approach. In secondary storage, objects are represented using the checkpoint scheme and maintained by a file server. Each

object is managed by a version manager which 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, and committed. When an action modifies an object, the object is opened and a copy of its current version is loaded in 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 re-opened 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.

A garbage collection scheme is supplied to periodically collect and destroy all dormant objects which are no longer referenced. This is the only method by which dormant objects can be deleted. The garbage collector runs only when there are no operational objects.

The explicit object scheduling scheme is supported and a totally visible distributed environment is provided. System calls can be made by a user to examine the load of any processor and to explicitly create his objects on specific processors.

Emerald

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

A local garbage collector and a distributed garbage collector are provided. Both collectors use a variation of the mark-and-sweep algorithm. The local garbage collector can be run at any time, independently of all the other workstations. To assist the collection procedure, each object maintains special bit which is set whenever a reference to the object is passed to another workstation. During the marking phase, the local collector checks this special bit to determine if

an object is referenced only by objects which reside on the same workstation. The distributed garbage collector requires the cooperation of all the workstations of the network. However, it does not require that all the workstations be functioning at the same time. As the workstations are recovered, the marking procedure will progress until all the workstations are searched. A detailed description of these garbage collecting schemes is given in [Jul 88].

An object migration scheme is supported. The task of moving objects is complicated even further because each object does not reside in its own address space. Consequently, when a large-grain object moves to a new workstation, and hence a new address space, all of its internal pointers have to be modified. Emerald solves this problem by generating re-locatable code and creating templates which describe the internal layout of the objects. Each template describes the internal structure of a particular object, including: pointers to objects which resides in it, and the data types of the objects. An object is moved according to the following procedure.

(a) All processes executing in the object are suspended.

(b) A template of the object's state is made.

(c) The object's state information and template are then sent to the new workstation.

(d) 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 of the object is traversed and the pointers are replaced with their new addresses.

(e) If the corresponding concrete type object (the code of the object) does not already exist at the workstation, it is obtained. The concrete type object and the state of the object are linked.

(f) Finally, the processes in the object are resumed.

Another problem that must be handled is moving the processes that were executing in an object which is migrated. Each object maintains a list of all processes which execute within it and the corresponding activation records which have to be moved. The system alters the stacks of these processes accordingly and moves the corresponding activation records to the new workstation so that the processes will execute properly. This may involve splitting a stack into multiple parts. In the worst case, the corresponding activation record is moved while those above it and those below it are separated into two new stacks. (See Figure 6.1). Details of this object migration scheme is given in [Jul 88].

The distributed environment is totally visible to the user.

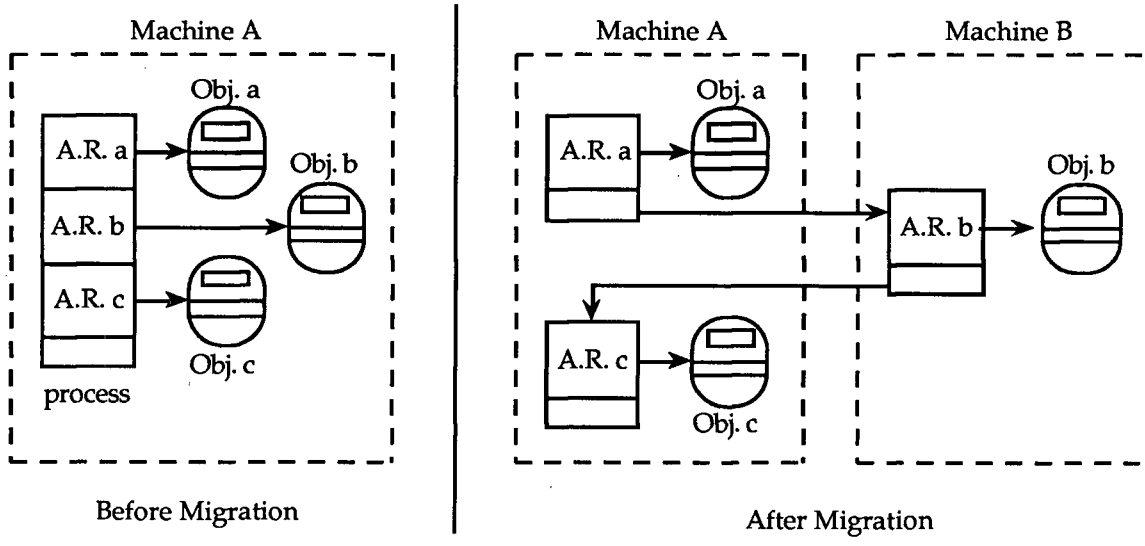

Figure 6.1: Object migration in Emerald

TABS/Camelot

TABS and Camelot provides three levels of storage: first, memory (or *volatile storage*) where objects reside when they are being accessed; second, secondary storage where objects reside

when they have not been accessed recently; third, stable storage where the log resides in order to survive failures.

Each object is represented in memory using the single version approach. A paging scheme is used. These systems differ from most DOBPSs in that an object can record its modifications to secondary storage before the corresponding action is committed. If an action aborts, its modifications are undone using the undo/redo log.

An object is represented in secondary storage using the undo/redo log scheme. TABS provides two types of log entries: old value/new value entries, and operation entries. Camelot also provides two types of log entries: old value/new value entries, and new value entries. An old value/new value entry records both the old and new values of a modified page of an object. A new value entry records the new value of a modified page of an object. An operation entry records the name of the operation invoked and enough information to invoke it again. Each action specifies the logging scheme to be used for the action. Periodically, each operating system performs a workstation-wide checkpoint of its objects to ensure a globally consistent state is maintained. To perform a checkpoint, the operating system informs all of the objects which reside in its workstation to suspend themselves in their next consistent state. When an object reaches a consistent state, it suspends its activities and notifies the operating system. When all objects have suspended themselves, a list of the pages which reside in memory and the status of all executing actions are recorded in the log. The activities of the system are then resumed.

Chapter 7.
An Example DOBPS

The design of a distributed, object-based programming system depends heavily on the intended application of the system. For example, a system designed for a distributed office environment will likely provide different features than a system designed for a distributed programming environment. In this chapter, we describe the design of a hypothetical DOBPS based on a specific hardware environment and on a particular set of goals.

7.1. Hardware Environment

- Collection of 8 SUN Workstations[15] interconnected by a 10Mbps Ethernet[16].
- Each workstation supplies a single Motorola M68020, 4MB of memory, special paging hardware, and a local 80MB hard disk.
- Processor pool of 24 M68020 processors, each with its own private memory.

---

15 SUN Workstation is a trademark of Sun Microsystems Inc.

16 Ethernet is a trademark of Xerox Corporation.

- A 400MB external disk. (See Figure 7.1).

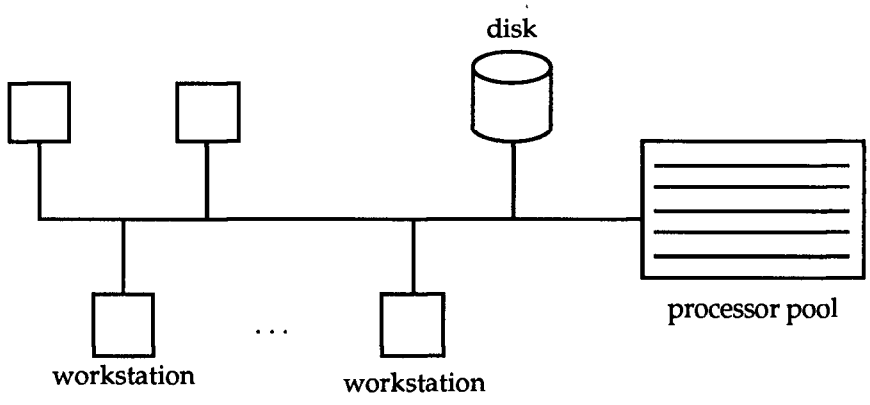

Figure 7.1: The hardware environment

7.2. Goals

- To provide 40 computer science graduate students with a distributed computing environment on which to create and execute distributed programs.
- To allow students to experiment with a DOBPS. Students should be permitted to examine and alter the operating system of a workstation. Consequently, the system should be as flexible as possible.
- A reasonably fast response time and high availability should be provided. A high degree of concurrency is not a major concern.
- If a workstation is separated from the network, it should be able to continue functioning. Furthermore, the rest of the system should not be affected by its loss.
- Users should be able to assign some of their objects to processors of the pool in order to execute them concurrently. Consequently, objects should be mobile.

7.3. Features

The operating system of the DOBPS should support the features described below.

Large and Medium-grain Objects

The large and medium-grain object model should be supported with each large-grain object residing in its own address space and encapsulating a number of medium-grain objects. For added protection, a medium-grain object should be invisible to clients outside of the large-grain object it resides in. That is, a medium-grain object may be invoked only by processes which reside in the same large-grain object.

This model provides a finer degree of control over data than the large-grain object model. Furthermore, it does not have the overhead associated with the large, medium and fine-grain object model. Providing a completely uniform object model is a nice luxury, but it is hardly necessary and it does not merit the extra overhead. Medium-grain objects should supply a fine enough degree of control to achieve the benefits of increased concurrency within the large-grain objects.

Active Object Model - dynamic variation

The dynamic variation of the active object model should be supported. This model provides additional protection by encapsulating objects and the processes that affect them. As a result, erroneous processes that may be created by a modified operating system can only affect the objects that they are restricted to. A single erroneous process cannot affect multiple objects, as is the case in the passive object model. The dynamic variation of this model also does not place a fixed limit on the number of invocations which can be handled concurrently by an object, as is the case in the static variation of this model. Consequently, deadlock should be less of a problem. To reduce the

overhead of dynamic process creation and destruction, a pool of idle processes should be maintained.

The passive object model should not be used mainly because of the expense of mapping processes into the address space of the objects they invoke. Maintaining a single address space for all objects and processes, such as the scheme used by Emerald, is not an appropriate solution either. This removes the hardware protection and security between the objects and eliminates two of the most important features of a DOBPS.

Nested Transactions

The nested transaction management scheme should be supported to guarantee that actions have the properties of serializability, atomicity, and permanence. This scheme has the added benefit that it permits the system to support the primitive used by distributed programs for concurrency. It also provides additional flexibility by enabling invocation failures to be tolerated. The transaction scheme does not provide this flexibility.

The request scheme provides the users with too much flexibility. The main problem with this scheme is that it does not guarantee that actions have the properties of permanence and atomicity. In a multiple user environment where objects may be shared, these are two very important features which should be provided by a DOBPS. The relatively small increase in performance does not justify the loss of these two features. If performance is an important goal, then the overhead of committing actions should be reduced.

The commit procedure should be coordinated using the fan-out approach, with each object affected by an action given the responsibility of performing the commit procedure on the objects it invokes. Object mobility is not complicated, as is the case when each operating system is responsible for handling the commit procedure, since each object maintains the information it

requires to perform the commit procedure. Furthermore, this approach does not have the difficulty of propagating the information needed to perform the commit procedure to the coordinator. This is not the case in the single coordinator approach.

Pessimistic Synchronization

A pessimistic synchronization scheme which uses read/write locks should be supported. This is a simple, efficient, and effective scheme for controlling the activities of multiple processes within an object while still permitting some degree of concurrency. A custom locking scheme which enables an object to define its own locking protocols, such as the schemes used by Clouds and TABS, should not be used because the system is not able to enforce action serializability.

Since a high degree of concurrency is not a major goal of this system, the optimistic synchronization scheme is not necessary. The main problem with this scheme is it may force some actions to abort and be redone. This is definitely an undesirable feature since a reasonably fast response time is wanted in this system. This scheme also introduces the added overhead of performing a serializability test and maintaining multiple versions of each object both in memory and in secondary storage.

User-managed Capabilities

The user-managed version of the capability scheme which uses encrypted capabilities should be supported. Security is a major issue in this system because students are permitted to experiment with and alter the operating system of a workstation. Therefore, the system cannot assume that all workstations may be trusted, as the system-managed and user & system-managed versions of this scheme do. Another drawback of these versions is that object mobility is complicated because capabilities are recorded by the system. Such is not the case in the user-managed capability scheme where the capabilities are maintained by the objects.

The added security provided by the control procedure scheme is probably not required in a university environment. Furthermore, the security required by the system does not justify forcing a programmer to supply a security scheme for each object. In contrast, the user-managed capability scheme provides a uniform, system-level security scheme for all objects.

Roll-back Object Recovery

This system represents objects in secondary storage using the checkpoint scheme (see below), therefore a roll-back object recovery scheme should be supported. This scheme has the benefit that a failed object can be restored to a consistent state very quickly and efficiently.

The drawback of the roll-back recovery scheme is the loss of those processes and invocations that were executing at the time of the failure. Recovering these activities using a roll-forward object recovery scheme is usually very complex and expensive, and there is no guarantee that an object will be restored to a consistent state. Therefore, a roll-forward recovery scheme should not be used unless a high degree of failure transparency is a design objective, which is not the case in our example.

Immutable Object Replication

This system should permit immutable objects to be replicated on multiple processors. This enables important system services to be replicated on each of the workstations so that a workstation can continue functioning even if it is separated from the rest of the network. For example, immutable objects such as compilers may be replicated on multiple workstations. This additional amount of read availability should provide a sufficient amount of availability for this system.

A primary copy or a peer copy replication scheme allows additional write availability to be provided. However, both of these schemes are complex and have a high overhead because they must deal with the problems of maintaining state consistency, controlling synchronization, and handling network failures. Consequently, these schemes should probably not be used since a reasonably fast response time is desired.

Message Passing

Object interactions should be handled using the message passing scheme because this system supports the active object model. This scheme has the advantage that it does not permit an objects to be accessed directly, thereby providing an extra degree of protection. A port scheme should also be supported so that messages are sent indirectly to objects via ports. This gives the system added flexibility because a tight binding is not created between messages and objects.

The direct invocation scheme cannot be used because of its incompatibility with the active object model.

The pass-by-reference scheme should be supported because it should have the lowest overhead of the three parameter passing schemes. The only objects which can be referenced are large-grain objects, since medium-grain objects cannot be observed outside of their large-grain objects. Therefore, the overhead of the pass-by-value scheme and the call-by-move scheme will be high due to the large amount of information which has to be transferred. Furthermore, the call-by-move scheme cannot be supported because an object migration scheme is not provided by this system (see below).

Cache/Broadcast Object Location

A cache/broadcast object location scheme should be supported to permit objects to be located quickly and efficiently. This scheme also permits objects to be mobile while avoiding the unnecessary expense and delay of having to notify other entities of the system whenever an object is moved. Such is not the case with the distributed name server scheme and the pure cache scheme. The broadcast portion of this scheme produces the most overhead; however, it should not be very large in this system because of the small size of the network.

The other object location schemes should not be used for a variety of reasons. The locations of the objects cannot be encoded in their identifiers because mobile objects are supported by this system. The distributed name server scheme and the cache location scheme both have the problem of a relatively high overhead in order to maintain accurate and up-to-date location information. The pure cache scheme also has the problem that each operating system may be forced to maintain a large cache. The pure broadcast scheme will likely be too slow and inefficient to satisfy the goal of fast response time required by this system.

This system is also probably too small to justify the use of forward location pointers. Furthermore, objects which do move can be located relatively efficiently in the broadcast portion of the scheme. Therefore, forward location pointers should not be used.

Clients - Invocation Probe

The invocation probe scheme should be supported to enable the clients to detect invocation failures. This scheme has the highest overhead of the three types of invocation failure detection schemes; however, it also best determines when an invocation has failed. To reduce the overhead of this scheme, a relatively large time-out period should be used to ensure that a server object is not disturbed too frequently.

The time-out scheme should not be used because it may lead to unnecessary failures if the time-out period is not determined properly. Determining the length of time to wait will be difficult in this type of environment. For example, some programs, such as simulations, may execute for longer periods of time than other programs, such as compilers.

The object probe scheme should not be used either because of it drawback that it simply checks the status of the server object and not the invocation. This may lead to the situation that a client waits indefinitely for a result that will never arrive. This is a highly undesirable characteristic for a system which is being experimented with.

Servers - Probe

The servers should support the probe scheme to detect invocation failures primarily because a probe scheme is supported by the clients and therefore no additional mechanism is required. The probe scheme has the benefit that it provides a simple and efficient way to detect the failure of a client.

The time-out scheme should not be used because it may lead to unnecessary failures. The main problem with the status report scheme is its possibly high overhead due to the large amount of information which may have to be maintained, passed, and processed by the operating systems of the network. Consequently, the goal of a fast response time will most likely not be satisfied when this scheme is used.

Secondary Storage - Checkpoint

Objects should be stored in secondary storage as efficiently as possible because of the limited amount of space available on the local hard disks. Therefore, the checkpoint scheme should be supported. A paging scheme should also be used to reduce the overhead of the checkpoint scheme.

As a result, objects will be composed of a set of pages which are mapped in and out of memory on demand, and only those pages which have been modified are written to secondary storage when an action commits.

It is not necessary to use the history of checkpoints scheme because this system supports a pessimistic synchronization scheme. Using a log scheme may reduce the amount of information that has to be written to secondary storage whenever an action commits; however, it also increases the expense of object recovery. Nevertheless, the history of checkpoints scheme and all the log schemes require much more storage space to be used than the checkpoint scheme. Consequently, these schemes should not be supported.

Memory - Single Version Approach

Objects should be represented in memory using the single version approach because the system supports a pessimistic synchronization scheme. Consequently, the multiple versions approach should not be used because it makes managing a pessimistic synchronization scheme more difficult.

This system also supports a nested transaction scheme; therefore, each version should be represented using either the stack of versions scheme or the version manager scheme. The stack of versions scheme should probably be used because it is slightly less complicated than the version manager scheme and it does not introduce additional objects which must be managed by the system. The undo/redo approach cannot be supported because this system represents objects in secondary storage using the checkpoint scheme.

Explicit Object Deallocation + Local Garbage Collector

An explicit object deallocation scheme and a local garbage collection scheme should be supported. Large-grain objects are explicitly deleted by their owner. The local garbage collector is

responsible for destroying medium-grain objects which are no longer referenced by other objects which reside in the same large-grain object. The garbage collector should also be responsible for periodically destroying the pool of idle processes to ensure that memory is not tied up unnecessarily.

Simply supporting an explicit object deallocation scheme is not sufficient because a user should not have to concern himself with destroying unwanted medium-grain objects. Simply supporting a garbage collection scheme is not sufficient either because it does not permit a user to have full control over his objects. For example, a user should be able to specify when an object is to be destroyed.

Processors - Explicit Object Scheduling

An explicit object scheduling scheme should be supported to permit users to control where objects are assigned. However, a restriction has to be placed on where objects may be located: a user's object must be assigned either to the processor of the user's workstation or to a processor of the pool, it cannot be assigned to another workstation. The reason for this restriction is to satisfy the goal of ensuring that the removal or failure of a workstation does not affect the other workstations or users of the network.

An implicit object scheduler should not be used because it does not permit a user to have full control over the activities of the system.

An object migration scheme should not be necessary in a system of this small a size. Furthermore, the modest increase in performance and availability achieved by the object migration scheme do not justify the expense, overhead, and problems of this scheme.

Totally Visible Distribution

The distribution should be totally visible to the users of the system and they should be permitted to have full control over the system's activities. This permits the users to experiment with the system. For example, a user should be able to specify whether the secondary storage versions of his objects will reside on his local disk or on an external disk.

Hiding the distribution or only supplying a partial view of the distribution greatly reduces the flexibility of a DOBPS and forces the system to provide those features related to distribution. This is not appropriate for a system which is being experimented with.

Chapter 8. Conclusion

8.1. Summary

In this thesis we defined a distributed object-based programming system as a distributed operating system which is designed to support the object abstraction featured by an object-based programming language. We then proceeded to characterize these types of operating systems by classifying them according to the features they provide. These features were separated into four categories: object structure, object management, object interaction management, and resource management. Objects are characterized by both their granularity and their composition.

Four features that DOBPSs support for managing objects were identified: first, an action management scheme that manages the activities of actions which affect multiple objects; second, a synchronization mechanism that controls the activities of multiple actions which invoke the same object simultaneously; third, a security mechanism that protects objects from unauthorized use; fourth, an object reliability scheme that enables objects to recover from failures in the system.

We identified three features that DOBPSs support for managing object interactions: first, a system-level invocation mechanism that handles object interactions; second, a location mechanism that determines the workstation in which an invoked object resides; third, an invocation failure detection mechanism that enables clients and servers to detect invocation failures.

Finally, four features that DOBPSs support for managing the physical resources of the network were identified: first, a memory management scheme that manages objects in memory; second, a secondary storage management scheme that manages objects in secondary storage; third, a process scheduler that assigns objects to processors and possibly moves them from one processor to another; fourth, a workstation management scheme that controls how the distributed environment of the system is viewed.

8.2. Future Research Directions

There are many features that may be supported by a DOBPS but whose issues need to be studied further, some are briefly described below.

Handling Network Failures

A DOBPS should be able to handle and recover from network failures. A network link failure may result in the network being divided into two separate groups, with workstations on one side of the partition unable to communicate with those on the other side. Most DOBPSs ignore the problem of dealing specifically with network failures because they rarely occur in a local area network. Instead, network failures are typically treated and handled as workstation failures. This is usually sufficient in most, but not all, cases.

Some of the problems cause by network partitions have been described earlier. In particular, when an object replication scheme (see §4.4.2.) is supported, a network failure may separate the

replicas of an object into two groups. If the replication scheme does not consider this problem, replicas on both sides of the partition may continue to accept and perform invocation requests from their section of the network. When the partition is repaired, an object may have two sets of replicas with inconsistent states. These differences have to be reconciled in order to restore the replicas to a consistent state.

Consequently, the problem of network failures cannot be totally ignored.

Heterogeneous Hardware

A DOBPS may allow the workstations of the network to support different types of processors. This gives a DOBPS added flexibility because it enables any available workstation to be added to the network in order to increase the processing power of the system. The main difficulty is handling communication between workstations which have heterogeneous processors. Different workstations may have different data representations or use different communication protocols. Therefore, a set of communication standards may have to be enforced.

When a heterogeneous environment is supported, object replication schemes, object scheduling schemes, and object migration schemes are more complex because they will have to select suitable processors.

Inheritance

A DOBPS may provide operating system level support for the concept of inheritance (see §2.2.). Unfortunately, implementing inheritance at this level is a difficult task. An object-oriented programming language may allow an object to directly examine and manipulate the state of its superclass. An operating system on the other hand, cannot allow this since the feature of

encapsulation would be violated and the security of the data would be compromised. An operating system should only permit the operations of a superclass to be inherited by an object.

The main problem consideration when implementing an inheritance scheme is determining the representation of the objects and class hierarchy.

Replicated Processes

A DOBPS may provide an action replication scheme along with an object replication scheme in order to increase the reliability of the actions. When an action is initiated, a number of action replicas are created. An action replica which makes an invocation on an object is given its own object replica to examine and manipulate. An object replica is associated with the same action replica throughout the lifetime of the action and cannot be invoked by any other action replica. When the first action replica successfully completes, the action may be committed.

There are at least two advantages to using a replicated process scheme. First, this scheme enables an action to proceed despite a number of transient failures. As long as one of the action replicas survives, the action survives. Second, this scheme can be used to reduce the probability of a non-deterministic software error. For example, the results of an action's replicas can be examined and a result selected using a majority scheme. This type of reliability and availability may be necessary for certain real-time systems.

There are also at least two problems associated with the action replication scheme. First, managing the replicated actions and the replicated objects is a very difficult task. For example, the system must ensure that each action replica invokes only those object replicas that are associated with it. Second, the system will experience a loss of throughput when actions are replicated. Furthermore, system resources are wasted if the results of an action replica are not wanted.

Wide Area Networks/Long-haul Networks

A DOBPS may be developed for a wide area network to enable a large number of workstations, connected over a long distance, to interact. Unfortunately, there are many problems associated with using wide area networks. First, the bridges or gateways connecting the local area networks introduce additional points of failure. Second, network partitions which occur in a wide area network may exist for a long period of time before being repaired. In contrast, network partitions which occur in a local area network can be repaired relatively quickly. Third, it is not practical to provide high levels of resource sharing because communication over a wide area network is still relatively slow and unreliable. Consequently, objects which interact frequently should be clustered, preferably on the same local area network. Finally, the task of determining the location of an invoked object is difficult. The pure cache scheme is impractical due the enormous amount of information that each operating system would have to maintain. A broadcast scheme is also impractical because wide area networks usually do not support a broadcast protocol, and the expense of sending a request message to each workstation of the network is usually substantial.

8.3. Concluding Remarks

Designing a distributed, object-based programming system is a very difficult and complex task. It is not possible at this time, nor does it seem likely in the future, that a single set of features can be declared as the optimal solution which should be supported by all DOBPSs. Many of the features which should be supported by a DOBPS depend on the intended application of the system, and the type of hardware resources which are to be joined by the DOBPS. At best, a set of features may be an optimal solution for a particular type of DOBPS, but even this may be subjective. In fact,

there is ample room for further studies as to which is the "best" solution to adopt for specific situations.

The development of a DOBPS is simplified by the use of objects. Objects serve well as the units for protection, recovery, security, synchronization, and mobility because they are autonomous entities.

The main 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 which reduces the complexity of the man-machine interface. In our opinion, this is the fundamental characteristic of these systems. A DOBPS simplifies the programming language interface to permit a programmer to express his ideas in a program conveniently. It also simplifies the operating system interface to enable a program to execute efficiently on a machine. This is a complete reversal of the past trend: instead of man complying to the demands of a machine, machines are now being built to comply to the demands of their users.

Bibliography

[Agrawala 82] A.K. Agrawala, S.K. Tripathi and G. Ricart, "Adaptive Routing using Virtual Time Techniques", *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 1, Jan. 1982.

[Ahamad 87a] M. Ahamad and P. Dasgupta, *Parallel Execution Threads: An Approach to Fault-Tolerant Actions*, Technical Report GIT-ICS-87/16, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, Mar. 1987.

[Ahamad 87b] M. Ahamad, P. Dasgupta, R.J. LeBlanc and C.T. Wilkes, "Fault Tolerant Computing in Object Based Distributed Operating Systems", *IEEE 6th Symposium on Reliability in Distributed Software and Database Systems*, Mar. 1987, pp. 115-125.

[Alsberg 76] P.A. Alsberg and J.D. Day, "A Principle for Resilient Sharing of Distributed Resources", *Proceedings of the 2nd International Conference on Software Engineering*, 1976, pp. 562-570.

[Almes 85] G.T. Almes, A.P. Black, E.D. Lazowska and J.D. Noe, "The Eden System: A Technical Review", *IEEE Transactions on Software Engineering*, SE-11(1), Jan. 1985, pp. 43-58.

[Andrews 88] G.R. Andrews, R.A. Olsson, M. Coffin, I. Elshoff, K. Nilsen, T. Purdin and G. Townsend, "An Overview of the SR Language and Implementation", *ACM Transactions on Programming Languages and Systems*, Vol. 10, No. 1, Jan. 1988, pp. 51-86.

[Banino 82] J.S. Banino and J.C. Fabre, "Distributed Coupled Actors: A CHORUS Proposal for Reliability", *IEEE 3rd International Conference on Distributed Computing Systems*, Oct. 1982, pp. 128-134.

[Banino 85] J.S. Banino, J.C. Fabre, M. Guillemont, G. Morisset and M.Rozier, "Some Fault-Tolerant Aspects of the CHORUS Distributed System", *IEEE 5th International Conference on Distributed Computing Systems*, May 1985, pp. 430-437.

[Bernstein 81] P.A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys*, Vol. 13, No.2, June 1981, pp. 185-221.

[Bernstein 84] P.A. Bernstein and N. Goodman, "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases", *ACM Transactions on Database Systems*, Vol. 9, No. 4, Dec. 1984, pp. 596-615.

[Birtwistle 73] G.M. Birtwistle, O.-J. Dahl, B. Myhrtag, and K. Nygaard, *Simula Begin*, Auerbach, Philadelphia, 1973.

[Black 85] A.P. Black, "Supporting Distributed Applications: Experience with Eden", *ACM Proceedings 10th Symposium on Operating System Principles*, Dec. 1985, pp. 181-193.

[Black 86a] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter, *Distribution and Abstract Types in Emerald*, Technical Report 86-02-04, Department of Computer Science, University of Washington, Seattle, WA, Feb. 1986.

[Black 86b] A. Black, N. Hutchinson, E. Jul, and H. Levy, *Object Structure in the Emerald System*, Technical Report 86-04-03, Department of Computer Science, University of Washington, Seattle, WA, April 1986.

[Booch 86] G. Booch, "Object-Oriented Development", *IEEE Transactions on Software Engineering*, Vol. SE-12, Num. 2, Feb. 1986, pp. 211-221.

[Bryant 81] R.M. Bryant and R.A. Finkel, "A Stable Distributed Scheduling Algorithm", *IEEE Proceedings of the 2nd International Conference on Distributed Computing Systems*, 1981, pp. 314-323.

[Cheriton 88] D. Cheriton, "The V Distributed System", *Communications of the ACM*, Vol. 31, No. 3, Mar. 1988, pp. 314-333.

[Chow 79] Y.C. Chow and W.H. Kohler, "Models for Dynamic Load Balancing in Heterogeneous Multiple Processor Systems", *IEEE Transactions on Computers*, Vol. C-28, No. 5, May 1979, pp. 354-361.

[Cohen 75] E. Cohen and D. Jefferson, "Protection in the Hydra Operating System", *Proceedings - 5th Symposium on Operating System Principles*, Vol. 9, No. 5, Nov. 1975, pp. 141-160.

[Cox 83] B.J. Cox, "Object-oriented programming in C", *Unix Review*, Oct./Nov. 1983.

[Cox 84] B.J. Cox, "Message/Object Programming: An Evolutionary Change in Programming Technology", *IEEE Software*, Vol. 1, No. 1, Jan. 1984, pp. 50-61.

[Dahl 66] O.-J. Dahl and K. Nygaard, "Simula - an ALGOL-Based Simulation Language", *Communications ACM*, Sept. 1966, Vol. 9, No. 9, pp. 671-678.

[Dasgupta 85] P. Dasgupta, R. LeBlanc and E. Spafford, *The Clouds Project: Designing and Implementing a Fault Tolerant Distributed Operating System*, Technical Report GIT-ICS-85/29, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1985.

[Dasgupta 86] P. Dasgupta, "A Probe-Based Monitoring Scheme for an Object-Oriented, Distributed Operating System", *ACM Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, 1986, pp. 57-66.

[Dasgupta 88] P. Dasgupta, R. LeBlanc and W. Appelbe, "The Clouds Distributed Operating System. Functional Description, Implementation Details and Related Work.", *IEEE 8th International Conference on Distributed Computing Systems*, San Jose, 1988.

[Diedrich 87] J. Diederich & J. Milton, "Experimental Prototyping in Smalltalk", *IEEE Software*, May 1987, pp. 50-64.

[Dijkstra 78] E.W. Dijkstra, et al., "On-the-fly garbage collection: An exercise in cooperation", *Communications of the ACM* , Vol. 21, No. 11, Nov. 1978, pp 966-975.

[DOD 80] *Ada Reference Manual*, U.S. Department of Defence, July 1980.

[Eppinger 85] J.L. Eppinger and A.Z. Spector, *Virtual Memory Management for Recoverable Objects in the TABS Prototype*, Technical Report CMU-CS-85-163, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, Dec. 1985.

[Gifford 79] D.K. Gifford, "Weighted Voting for Replicated Data", *ACM Proceedings of the 7th Symposium on Operating System Principles*, Dec. 1979, pp. 150-162.

[Goldberg 83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Massachusetts, 1983.

[Gray 78] J.N. Gray, "Notes on data base operating systems", In *Lecture Notes in Computer Science*. G. Goos and J. Hartmanis, Springer-Verlag, New York, 1978, pp. 393-481.

[Gray 80] J.N. Gray, *A Transaction Model*, Technical Report RJ2895, IBM Research Laboratory, San Jose, California, Aug. 1980.

[Gray 81] J.N. Gray, et al., "The Recovery Manager of the System R Database Manager", *ACM Computing Surveys*, Vol. 13, No. 2, June 1981, pp. 223-242.

[Guillemont 87] M. Guillemont and J.L. Martins, "CHORUS: a new UNIX for the distribution age", Paper submitted for publication. Currently available from the authors at INRIA, Feb. 1987.

[Herlihy 86] M.P. Herlihy, "Optimistic Concurrency Control for Abstract Data Types", *5th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Aug. 1986, pp. 206-217.

[Hutchinson 87] N.C. Hutchinson, R.K. Raj, A.P. Black, H.M. Levy, and E. Jul, *The Emerald Programming Language*, Technical Report 87-10-07, Department of Computer Science, University of Washington, Seattle, WA, Oct. 1987.

[Jessop 82] W.H. Jessop, J.D. Noe, D.M. Jacobson, J.-L. Baer and C. Pu, "The Eden Transaction-Based File System", *IEEE Proceedings 2nd Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, PA., July 1982, pp. 163-169.

[Jones 75] A.K. Jones and W.A. Wulf, "Towards the design of secure systems", *Software - Practice and Experience*, Oct./Dec. 1975, Vol. 5, No. 4, pp. 321-336.

[Jones 76] A.K. Jones, "The Narrowing Gap Between Language Systems and Operating Systems", *Computer Science Research Review 1975-1976 Carnegie-Mellon University*, pp. 17-23.

[Jones 84] T.C. Jones, "Reusability in Programming: A survey of the State of the Art", *IEEE Transactions on Software Engineering*, September 1984, pp.488-494.

[Jones 86] M.B. Jones and R.F. Rashid, "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Systems", *ACM Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, 1986, pp. 67-77.

[Jul 88] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-Grained Mobility in the Emerald System", *ACM Transactions on Computer Systems*, Vol. 6, No. 1, Feb. 1988, pp. 109-133.

[Kahn 81] K.C. Kahn, W.M. Corwin, T.D. Dennis, H. D'Hooge, D.E. Hubka, L.A. Hutchins, J.T. Montague, F.J. Pollack, and M.R. Gifkins, "iMAX: A Multiprocessor Operating System for an Object-Based Computer", *Proceedings of the 8th Symposium on Operating System Principles*, Vol. 15, No. 5, Dec. 1981, pp. 127-136.

[Korth 83] H.F. Korth, "Locking Primitives in a Database System", *Journal of the ACM*, Vol. 30, No. 1, Jan. 1983, pp. 55-79.

[Lazowska 81] E.D. Lazowska, H.M. Levy, G.T. Almes, M.J. Fischer, R.J. Fowler and S.C. Vestal, "The Architecture of the Eden System", *ACM Proceedings 8th Symposium on Operating System Principles*, Dec. 1981, pp. 148-159.

[Liskov 83a] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, July 1983, pp. 381-404.

[Liskov 83b] B. Liskov and M. Herlihy, "Issues in Process and Communication Structure for Distributed Programs", *3rd Symposium on Reliability in Distributed Software and Data Base Systems*, Oct. 1983, pp. 123-132.

[Liskov 87] B. Liskov, D. Curtis, P. Johnson and R. Scheifler, "Implementation of Argus", *ACM Proceedings 12th Symposium on Operating System Principles*, 1987, pp. 111-122.

[Liskov 88] B. Liskov, "Distributed Programming in Argus", *Communications of the ACM*, Vol. 31, No. 3, March 1988, pp. 300-312.

[McKendry 85] M.S. McKendry and M. Herlihy, *Time Driven Orphan Elimination*, Technical Report CMU-CS-85-138, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, July 1985.

[MIT 86] *Laboratory for Computer Science Progress Report 23*, Massachusetts Institute of Technology, 1986.

[Moss 85] J.E. Moss, *Nested transactions: An approach to reliable distributed computing*, Technical Report MIT/LCS/TR-260, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1985.

[Mullender 85] S.J. Mullender and A.S. Tanenbaum, "A Distributed File Service Based on Optimistic Concurrency Control", *ACM 10th Symposium on Software Principles*, 1985.

[Mullender 86] S.J. Mullender and A.S. Tanenbaum, "The Design of a Capability-Based Distributed Operating System", *The Computer Journal*, Vol. 29, No. 4, Aug. 1986.

[Nicol 87] J.R. Nicol, G.S. Blair, and J. Walpole. "Operating System Design: Towards a Holistic Approach?", *ACM Operating Systems Review*, Vol. 21, No. 1, Jan. 1987, pp. 11-19.

[Notkin 87] D. Notkin, N. Hutchinson, J. Sanislo and M. Schwartz, "Heterogeneous Computing Environments: Report on the ACM SIGOPS Workshop on Accommodating Heterogeneity", *Communication of the ACM*, Vol. 30, No. 2, Feb. 1987, pp. 132-140.

[O'Shea 86] T. O'Shea, K. Beck, D. Halbert & K. Schmucker, "Panel: The Learnability of Object-Oriented Programming Systems", *ACM Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, Sept. 1986, pp. 502-504.

[Oki 85] B.M. Oki, B.H. Liskov and R.W. Scheifler, "Reliable Object Storage to Support Atomic Actions", *ACM Proceedings of the 10th Symposium on Operating System Principles*, 1985, pp. 147-159.

[Pitts 88] D.V. Pitts and P. Dasgupta, "Object Memory and Storage Management in the Clouds Kernel", *IEEE 8th International Conference on Distributed Computing Systems*, San Jose, 1988.

[Powell 83] M.L. Powell and D.L. Presotto, "Publishing - A Reliable Broadcast Communication Mechanism", *ACM Operating System Review*, Vol. 17, No. 5, pp. 100-109.

[Pu 86] C. Pu, J.D. Noe, A. Proudfoot, "Regeneration of Replicated Objects: A Technique and Its Eden Implementation", *IEEE Proceedings 2nd International Conference on Data Engineering*, Feb. 1986, pp. 175-187.

[Rentsch 82] T. Rentsch, "Object Oriented Programming", *ACM SIGPLAN Notices*, Vol. 17, No. 9, Sept. 1982, pp. 51-57.

[Rozier 87] M. Rozier and J.L. Martins, "The CHORUS Distributed Operating System: Some Design Issues", *Distributed Operating Systems. Theory and Practice*, Springer-Verlag, Berlin, Heidelberg, 1987, pp. 262-287.

[Schwarz 84] P.M. Schwarz and A.Z. Spector, "Synchronizing Shared Abstract Types", *ACM Transactions on Computer Systems*, Vol. 2, No. 3, Aug. 1984, pp. 223-250.

[Sheth 86] A.P. Sheth and M.T. Liu, "Integrated Locking and Optimistic Concurrency Control in Distributed Database Systems", *IEEE Proceedings of the 6th International Conference on Distributed Computing Systems*, May. 1986, pp. 89-99.

[Smith 79] R.G. Smith, "The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver", *IEEE Proceedings of the 1st International Conference on Distributed Computing Systems*, 1979, pp. 185-192.

[Soltis 79] F.G. Soltis and R.L. Hoffman, "Design Considerations for the IBM System/38", *IEEE COMPCON S'79*, 1979, pp. 132-137.

[Spafford 84] E.H. Spafford and M.S. McKendry, *Kernel Structures for Clouds*, Technical Report GIT-ICS-84/09, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1984.

[Spafford 87] E.H. Spafford, *Object Operation Invocation in Clouds*, Technical Report GIT-ICS-87/14, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1987.

[Spector 84] A.Z. Spector, J. Butcher, D.S. Daniels, D. Duchamp, J.L. Eppinger, C.E. Fineman, A. Heddaya, and P.M. Schwarz, "Support for Distributed Transactions in the

TABS Prototype", *IEEE 4th Symposium on Distributed Software and Data Base Systems*, 1984, pp. 186-205.

[Spector 85] A.Z. Spector, D.S. Daniels, D. Duchamp, J.L. Eppinger, and R. Pausch, *Distributed Transactions for Reliable Systems*, Technical Report CMU-CS-85-117, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, Sept. 1985.

[Spector 86] A.Z. Spector, J.J. Bloch, D.S. Daniels, R.P. Draves, D. Duchamp, J.L. Eppinger, S.G. Menees, and D.S. Thompson, *The Camelot Project*, Technical Report CMU-CS-86-166, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, Nov. 1986.

[Spector 87a] A.Z. Spector, *Distributed Transaction Processing and the Camelot System*, Technical Report CMU-CS-87-100, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, Jan. 1987.

[Spector 87b] A.Z. Spector, D.S. Thompson, R.F. Pausch, J.L. Eppinger, D. Duchamp, R.P. Draves, D.S. Daniels, and J.J. Bloch, *Camelot: A Distributed Transaction Facility for Mach and the Internet - An Interim Report*, Technical Report CMU-CS-87-129, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, June 1987.

[Stankovic 84] J.A. Stankovic and I.S. Sidhu, "An Adaptive Bidding Algorithm for Processes, Clusters and Distributed Groups", *IEEE Proceedings of the 4th International Conference on Distributed Computing Systems*, 1984, pp. 49-59.

[Stone 77] H.S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms" *IEEE Transactions on Software Engineering*, Vol. SE-3, May 1977, pp. 315-321.

[Stone 78] H.S. Stone and S.H. Bokhari, "Control of Distributed Processes", *IEEE Computer*, Vol. 11, No. 7, July 1978, pp. 97-106.

[Stroustrup 86] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Massachusetts, 1986.

[Tanenbaum 81] A.S. Tanenbaum and S.J. Mullender, "An Overview of the Amoeba Distributed Operating System", *ACM Operating Systems Review*, Vol.15, No. 3, July 1981.

[Tanenbaum 85] A.S. Tanenbaum and R.V. Renesse, "Distributed Operating Systems", *Computing Surveys*, Vol. 17, No. 4, Dec. 1985, pp. 419-470.

[Tanenbaum 86] A.S. Tanenbaum, S.J. Mullender and R. van Renesse, "Using Sparse Capabilities in a Distributed Operating System", *IEEE Proceedings of the 6th International Conference on Distributed Computing Systems*, May 1986, pp. 558-563.

[Tanenbaum 87] A.S. Tanenbaum and R. van Renesse, "Reliability Issues in Distributed Operating Systems", *IEEE 6th Symposium on Reliability in Distributed Software and Data Base Systems*, March 1987.

[Tripathi 86] S.K. Tripathi and S. Huang, "Distributed Resource Scheduling for a Large Scale Network of Processors: HCSN", *IEEE Proceedings of the 6th International Conference on Distributed Computing Systems*, May 1986, pp. 321-327.

[Van Tilborg 81] A.M. Van Tilborg and L.D. Wittie, "Wave Scheduling: Distributed Allocation of Task Forces in Network Computers", *IEEE Proceedings of the 2nd International Conference on Distributed Computing Systems*, 1981, pp. 337-347.

[Walker 83] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System", *ACM Proceedings of the 9th Symposium on Operating Systems Principles*, Oct. 1983, pp. 49-70.

[Walker 84] E.F. Walker, *Orphan Detection in the Argus System*, Technical Report MIT/LCS/TR-326, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, June 1984.

[Wegner 87] P. Wegner, "Dimensions of Object-Based Language Design", *ACM Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, Oct. 1987, pp. 168-182.

[Wirth 85] N. Wirth, *Programming in Modula-2*, 3rd edition, Springer-Verlag, New York, New York, 1985.

[Wittie 80] L.D. Wittie and A.M. Van Tilborg, "MICROS, A Distributed Operating System for MICRONET, a Reconfigurable Network Computer", *IEEE Transactions on Computing*, Vol. C-29, Dec. 1980, pp. 1133-1144.

[Wulf 74] W.A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "HYDRA: The Kernel of a Multiprocessor Operating System", *Communications ACM*, June 1974, Vol. 17, No. 6, pp. 337-345.

[Zeigler 81] S. Zeigler, N. Allegre, D. Coar, R. Johnson, J. Morris, and G. Burns, "The Intel 432 Ada Programming Environment", *IEEE COMPCON S'81*, 1981, pp. 405-410.

[Zhou 85] S. Zhou and R. Zicari, *Object Management in Local Distributed Systems*, Technical Report UCB/CSD 86/267, Computer Science Division (EECS), University of California, Berkeley, CA, Nov. 1985.

Appendix A
Classification Scheme Overview

Abcdef	=	Category or feature
Abcdef	=	Implementation scheme
———	=	Alternate implementation schemes
┌ - - -┐ └ - - -┘	=	Variations of an implementation scheme
*	=	Multiple implementation schemes supported
(R.S.C.)	=	The scheme supported in the example

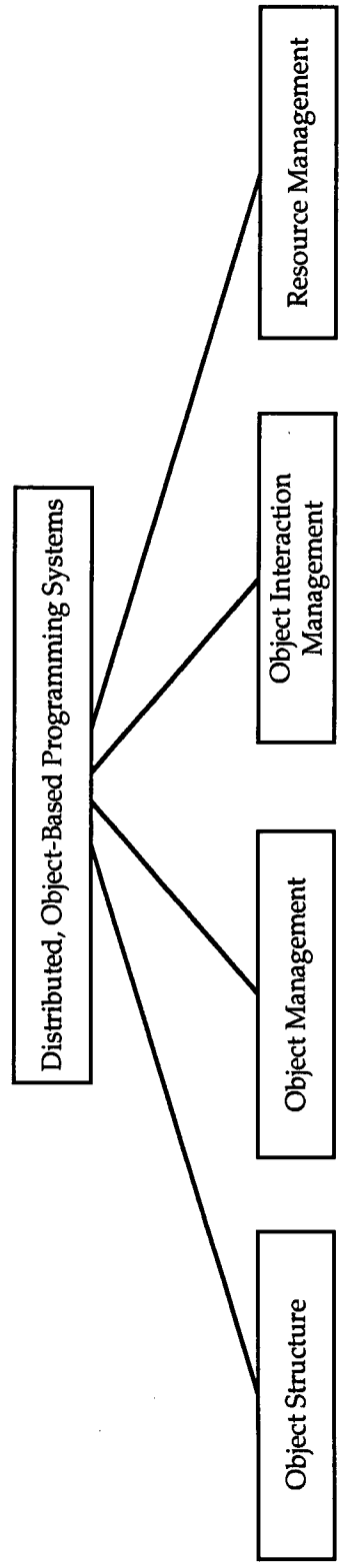

Figure 9.1: Categories of the classification scheme

Object Structure

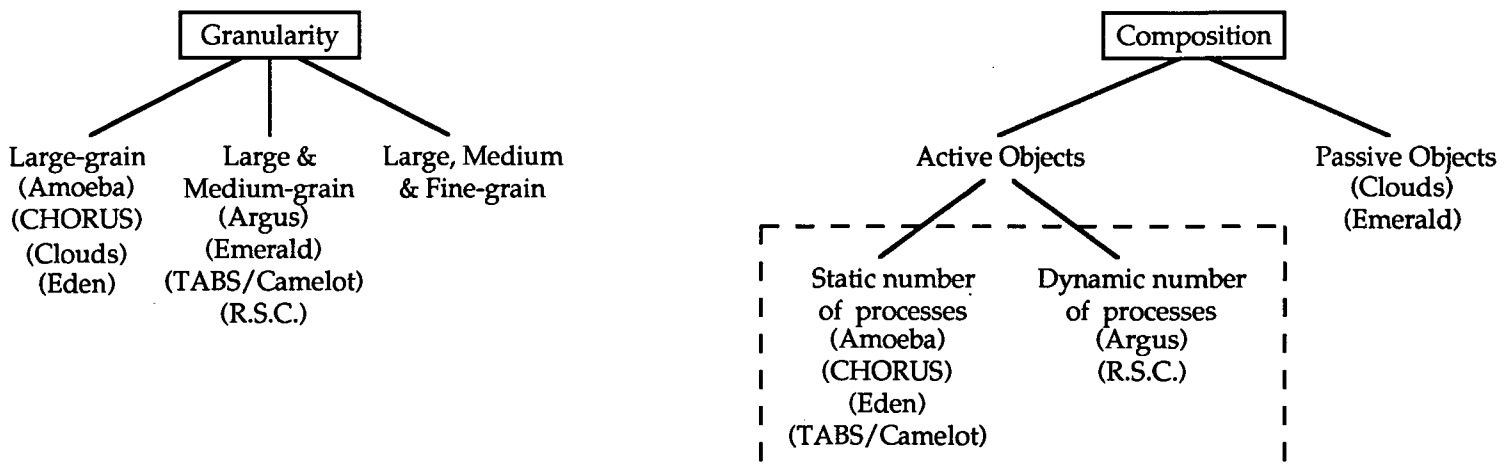

Figure 9.2: Object Structure

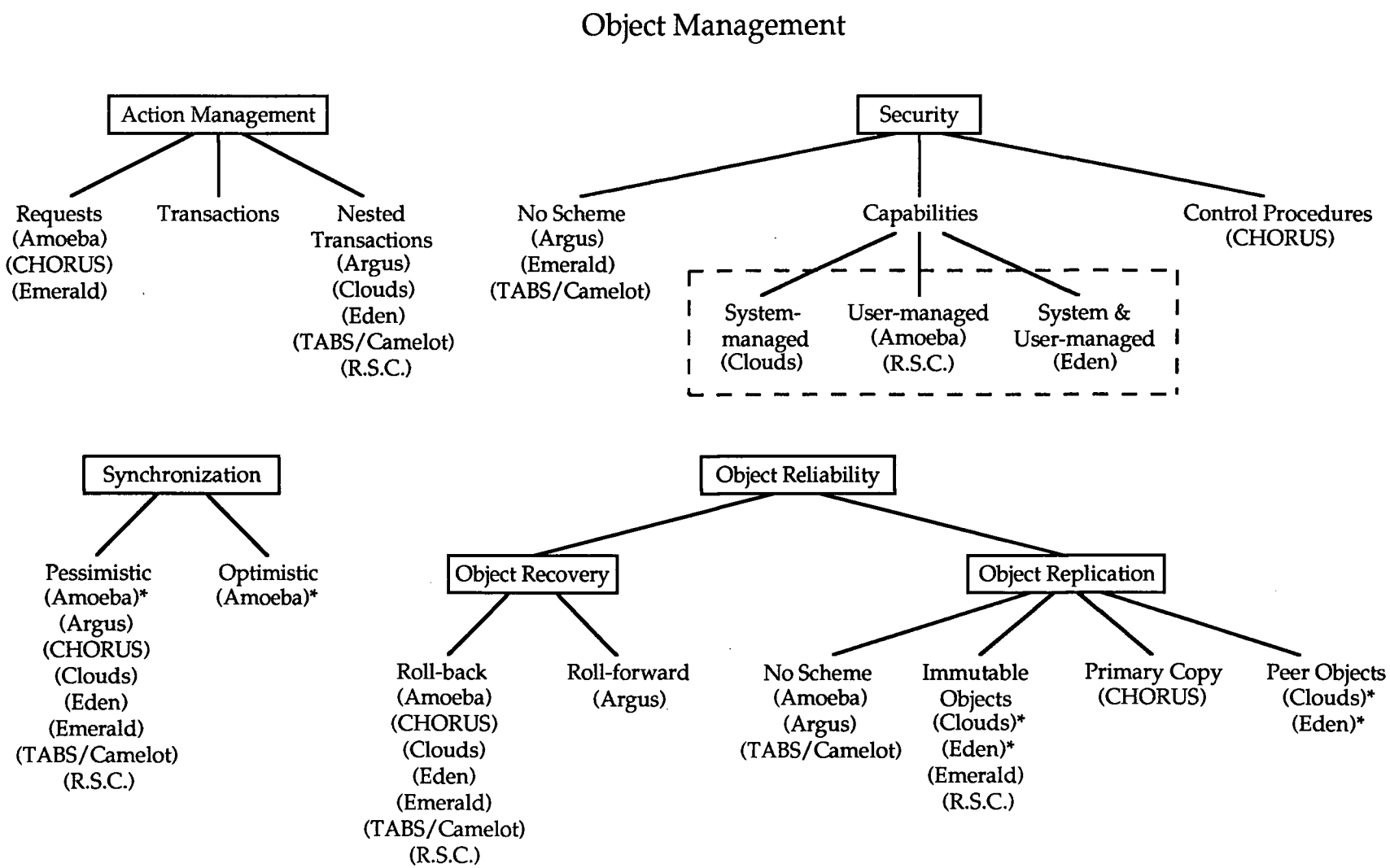

Figure 9.3: Object Management

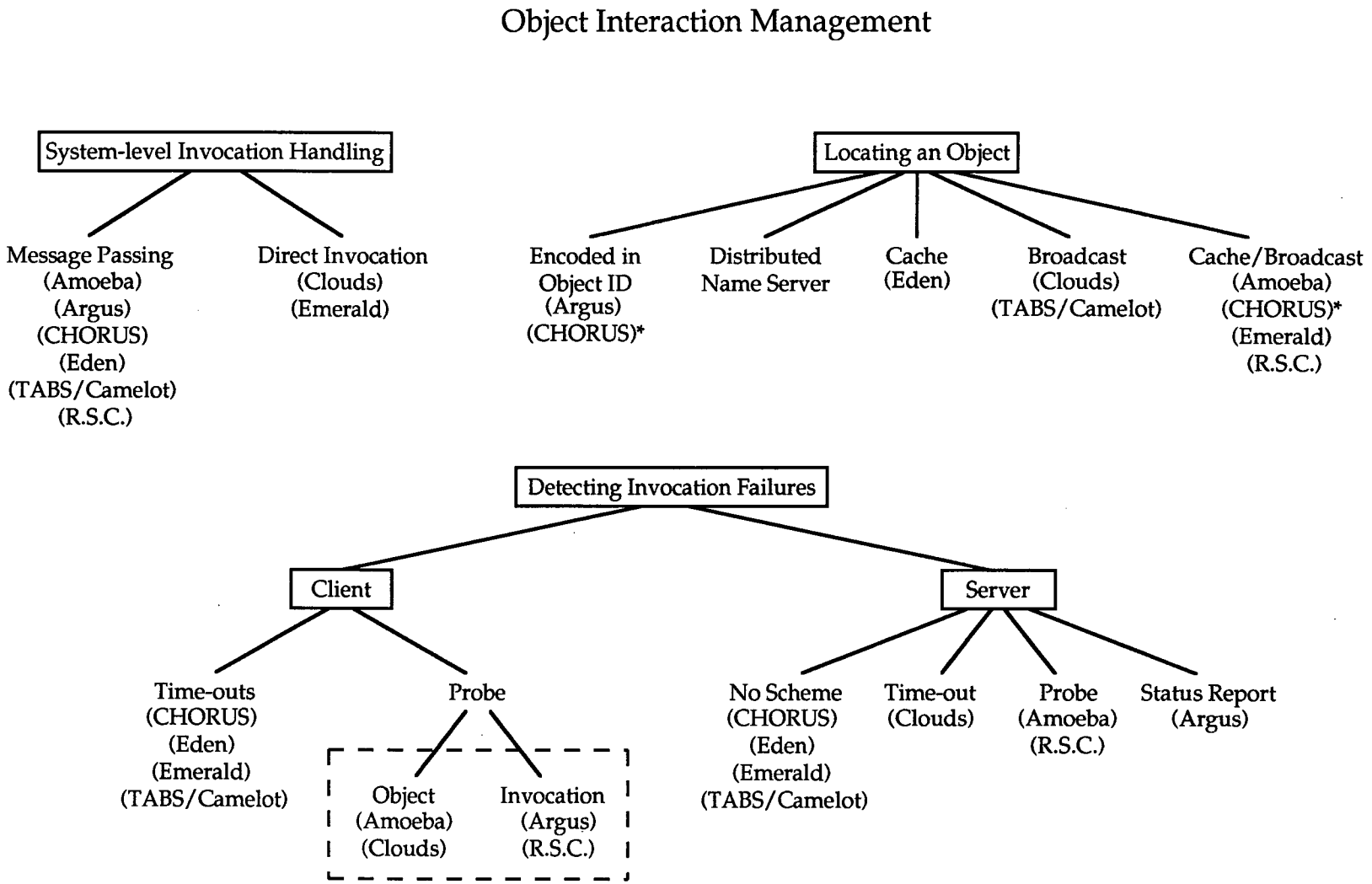

Figure 9.4: Object Interaction Management

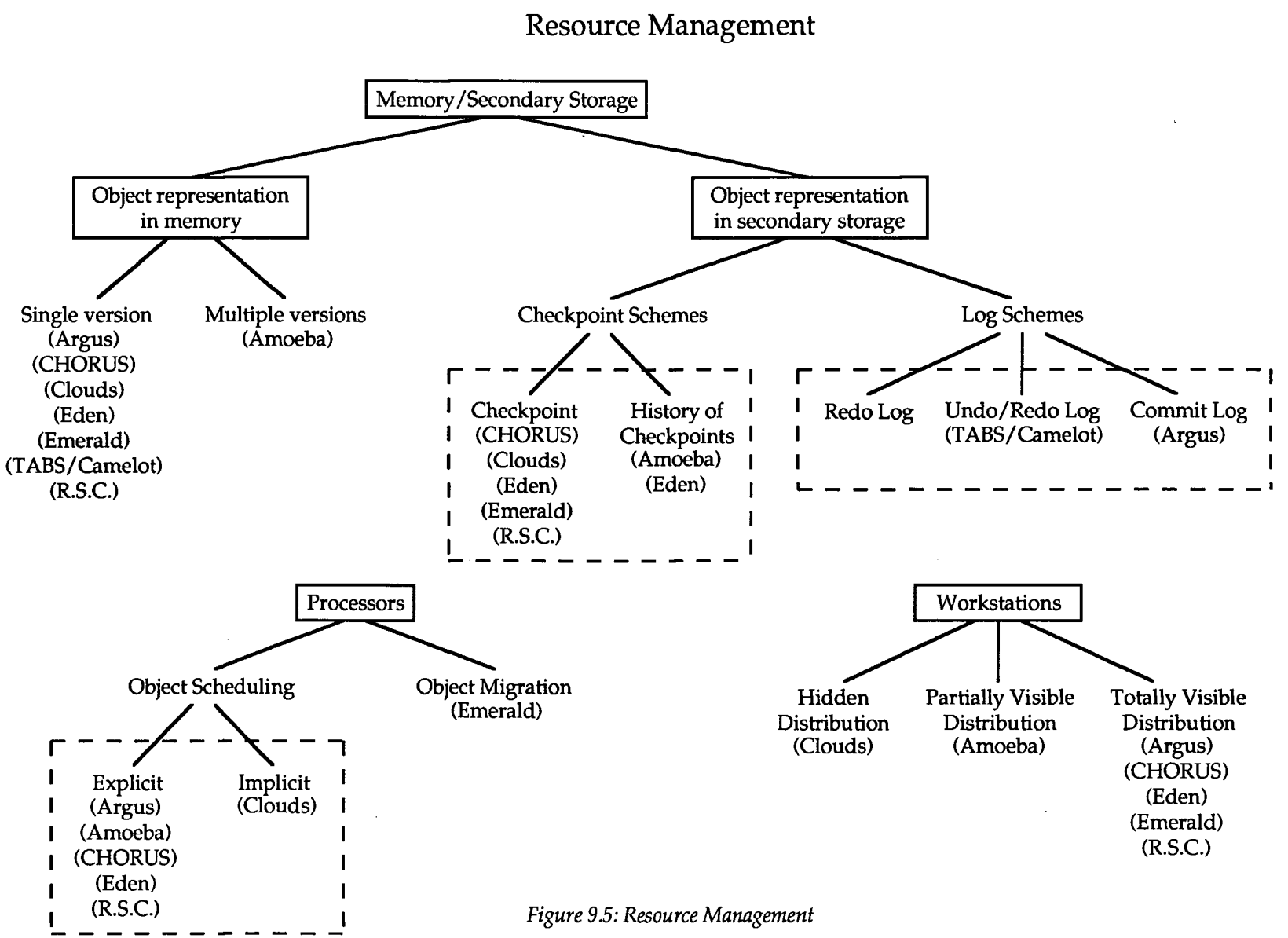

Figure 9.5: Resource Management

Appendix B
Feature Tables

The following two tables summarize the features which should and should not be supported together by a DOBPS. The vertical column indicates the primary feature supported by a DOBPS while the horizontal column indicates the secondary feature being considered.

	Large-grain Objects L & Medium-grain L, M & Fine-grain	Active Objects Passive Objects	Requests Transactions Nested Transactions	Pessimistic Synch. Optimistic Synch.	Capabilities Control Procedures	Roll-back Recovery Roll-forward Recov.	Immutable Objects Primary Copy Peer Objects	Message Passing Direct Invocation	Encoded in Object ID Dist. Name Server Cache Broadcast Cache/Broadcast	Client - Timeouts Client - Probes	Server - Timeouts Server - Probes Server - Status Reports	Memory - Single Vers. Memory - Mult. Vers.	S. Storage - Checkpoint S. S. - Hist. of Check. S. S. - Log	Explicit Object Sched. Implicit Object Sched. Object Migration	Hidden Distribution Partially Visible Totally Visible
Large-grain Objects L & Medium-grain L, M & Fine-grain		x √						x √							
Active Objects Passive Objects	x							√ x x √							
Requests Transactions Nested Transactions															
Pessimistic Synch. Optimistic Synch.												√ x x √	x √		
Capabilities Control Procedures															
Roll-back Recovery Roll-forward Recov.													x x √		
Immutable Objects Primary Copy Peer Objects															
Message Passing Direct Invocation	x	√ x x √													

Table I: Features of a DOBPS 1

	Large-grain Objects L & Medium-grain L, M & Fine-grain	Active Objects Passive Objects	Requests Transactions Nested Transactions	Pessimistic Synch. Optimistic Synch.	Capabilities Control Procedures	Roll-back Recovery Roll-forward Recov.	Immutable Objects Primary Copy Peer Objects	Message Passing Direct Invocation	Encoded in Object ID Dist. Name Server Cache Broadcast Cache/Broadcast	Client - Timeouts Client - Probes	Server - Timeouts Server - Probes Server - Status Reports	Memory - Single Vers. Memory - Mult. Vers.	S. Storage - Checkpoint S. S. - Hist. of Check. S. S. - Log	Explicit Object Sched. Implicit Object Sched. Object Migration	Hidden Distribution Partially Visible Totally Visible
Encoded in Object ID Dist. Name Server Cache Broadcast Cache/Broadcast														x x	
Client - Timeouts Client - Probes											x x √				
Server - Timeouts Server - Probes Server - Status Reports										x x √				x	
Memory - Single Vers. Memory - Mult. Vers.				√ x x √											
S. Storage - Checkpoint S. S. - Hist. of Check. S. S. - Log				x √		√ x √ x									
Explicit Object Sched. Implicit Object Sched. Object Migration									x x		x				x √ √
Hidden Distribution Partially Visible Totally Visible														x √ √ x	

Table II: Features of a DOBPS 2