Reliability Issues in the Design of Distributed Object-Based Architectures

by

Luigi Vincenzo Mancini

Ph.D. Thesis

The University of Newcastle upon Tyne
Computing Laboratory

March 1989

Abstract

This thesis is aimed at enhancing the existing set of techniques for building distributed systems, specifically from the point of view of fault-tolerant computing.

Reliability is of fundamental importance in the design and operation of distributed systems, as an increasing number of computers are employed in the automation of various essential services. In the past decade, much research effort has been concerned with the object-based methodology for the design and implementation of reliable distributed systems.

This thesis describes three contributions to this effort. First, it is shown that object-based programming features can in fact be introduced into procedural languages provided that these languages are endowed with certain facilities. Then, work is discussed which illustrates the relationship between distributed object-based architectures and an apparently different form of distributed architectures based on processes. This work puts the notion of object-based architectures into a new perspective, which shows that the object-based philosophy and the process-based philosophy are the dual of each other.

Finally, an important aspect of the design of an object-based distributed architecture is investigated, that of automatic garbage collection. A distributed garbage collection scheme is described that handles fault tolerance by an extension of the technique commonly employed to detect unwanted computations in distributed architectures. The scheme proposed can also be seen as yet a further illustration of the link between object-based and process-based architectures.

Acknowledgements

I owe a special debt of gratitude to my supervisor, Brian Randell, for suggesting my research topic, for his continuous and useful encouragement and precise hints, and for reading and commenting on the many drafts of this thesis.

I am also grateful to Santosh Shrivastava whose invaluable advice influenced my work on the duality of fault-tolerant system structures and on distributed garbage collection, and to Pete Lee for his constructive criticism in reading the crucial parts of this thesis.

Moreover, I must thank several colleagues at the Computing Laboratory for their advices and comments. In particular, I wish to acknowledge the many helpful discussions with Ron Kerr, Lindsay Marshall, Giuseppe Pappalardo, and Geppino Pucci.

I would also like to express my gratitude to Chiara, my wife, for being so patient throughout the difficult years of my research, and giving me great emotional support and companionship.

Financial support for this work was provided by grants from the Royal Signals and Radar Establishment of the U.K. Ministry of Defence, and from the Italian Consiglio Nazionale delle Ricerche.

Declaration

I declare that no part of this thesis has previously been submitted for any other degree or qualifications. Some material presented in this thesis has been published as detailed in the reference list.

Table of Contents

1. Introduction 1

1.1. Object-based programming 3

1.2. Distributed architectures 5

1.3. Structure and aims of the thesis 9

2. Object-Based and Object-Oriented Programming 12

2.1. Definitions and examples 13

2.1.1. Objects and classes 13

2.1.2. Sub-classes and inheritance 17

2.2. A classification 23

2.3. Concluding remarks 28

3. Object-Based versus Procedure-Based Programming Languages . 29

3.1. Motivations 30

3.2. Class definition 32

3.3. Sub-classes and inheritance 36

3.4. Sub-classing implementation exploiting polymorphic procedures 40

3.5. Sub-classing implementation at objects creation-time 45

3.6. Preprocessor specification 51

3.7. Performance considerations 55

3.8. Concluding remarks 56

4. Distributed Object-Based Architectures 60

4.1. Distribution issues 60

4.2. Access issues 64

4.2.1. Remote procedure calls 65

4.3. Failure issues 69

4.4. Concurrency issues 73

4.5. Migration and reconfiguration 76

4.6. Concluding remarks 77

5. Object-Based versus Process-Based Distributed Architectures ... 79

5.1. The canonical architectures 80

5.1.1. The object-based architecture 81

5.1.2. The process-based architecture 83

5.2. The duality argument 87

5.3. Some examples 95

5.4. An application of the duality mapping 99

5.5. A review of object replication techniques 100

5.5.1. The available copies scheme 101

5.5.2. Recovery 102

5.5.3. Read optimization 104

5.5.4. The primary copy scheme 106

5.6. Process replication techniques 108

5.6.1. The available processes scheme 110

5.6.2. Read optimization 110

5.6.3. The primary process scheme 111

5.7. Some existing process replication schemes 112

5.8. Concluding remarks 114

6. An Example of Object-Based Distributed System Design: Fault-Tolerant Garbage Collection 118

6.1. Notes on garbage collection 119

6.2. Object-based garbage collection and reliability requirements ... 125

6.3. RPCs and orphan detection and killing 132

6.4. Fault-tolerant garbage collection 134

6.4.1. Treatment of node failures 135

6.4.2. Reliable transfer of remote capabilities 140

6.4.3. Treatment of stable objects 142

6.4.4. Inter-node cycles 144

6.4.5. Performances 150

6.5. Concluding remarks 152

7. Conclusions 154

7.1. Summary of the thesis 154

7.2. Future work 159

References 162

Chapter 1

Introduction

Technical advances in large scale integration and in interconnection media have made distributed processing economically feasible. Technological and economic factors have combined to make *distributed systems* the most attractive and effective solutions for a large variety of applications. From relatively simple applications, in which a main computer makes the most important decisions, to more sophisticated applications, in which functionality is more uniformly dispersed, the notion of distributed systems is so appealing that it is often hard to justify alternative approaches.

Distributed systems and communication services are increasingly moving from a supportive role to an essential one in many commercial, industrial, educational and research organizations. It is widely recognized that the present trend towards distributed systems will also continue in the future. Taking advantage of continuing improvements in hardware cost and performance, there will be a continued tendency to put even more processing and storage power locally, and to interconnect machines with one another and with specialized servers to form large scale distributed systems.

However, in spite of the relatively long history of distributed systems (at least in computing terms), and the many successful (and unsuccessful) systems that have been built, there remain fundamental research issues that seem to be very resistant to solution. Many of the fundamental unresolved issues can be characterized as operating systems issues, as they deal with traditional operating system problems such as resource management and virtualization of machine characteristics into more convenient or understandable forms. For example, a particularly difficult problem for distributed systems is to ensure that the system conforms to the specification of its behaviour, or at least approximates its specification with some predictable degree of success. Obviously, one should try to make systems as correct as possible, but even perfect software will not act properly if the hardware refuses to work - note that the greater the number of computers in a distributed system, the higher the probability that one of them has crashed. As more and more computers are used in the automation of various essential services, the reliability of distributed systems becomes increasingly important, and efforts must be made to provide for a system reliability level significantly greater than that of the probability of all hardware behaving in a non-faulty fashion.

Many research projects have been using an *object-based* methodology for the design of reliable distributed systems. The work in this thesis employs the object-based model too, and covers many aspects of distributed system design including operating systems and programming languages. In each of these areas, the relationships to other programming methodologies is explored with the intention to put the notion of object-based system into a new perspective. The rest of this introductory chapter describes the object-based programming methodology and the notion of distributed architecture.

1.1. Object-based programming

Making software correct is easier said than done since, unfortunately, Dijkstra's prophecy has not yet come true ([Dijkstra72], p. 863):

> *"As a matter of fact, I think that we have learned so much that within a few years programming can be an activity vastly different from what it has been up till now, so different that we had better prepare ourselves for the shock. . . . The vision is that, well before the seventies have run to completion, we shall be able to design and implement the kind of systems that are now straining our programming ability at the expense of only a few percent in man-years of what they cost us now, and that besides that, these systems will be virtually free of bugs."*

Managing the complexity of software systems is still regarded as one of the key problems in Computer Science. The computer industry is still facing a *software crisis* - ambitions regarding quality and sophistication of systems, are continuing to outstrip the ability to design, implement, modify and enhance complex software systems in a reliable and cost-effective fashion. Nowadays, therefore, a crucial issue for the computer industry is the development of *architectures* that support, in a cost-effective fashion, the design, implementation, maintenance and evolution of complex systems.

A design methodology, known as *object-based,* has become popular in recent years. Object-based architectures are regarded by many as holding the solution of this problem in that they promote a new way for system developers to work together, resulting in an effective means of tackling complex issues along with providing potentiality for new software tools.

The fundamental characteristic of object-based architectures is that the entities that are manipulated at run-time by programs are *objects,* where an object is an encapsulation of some data together with the set of operations that are permitted on that data. At the programming language level, the mechanism for providing this encapsulation is an *abstract data type* mechanism [Liskov77], where the abstract data type describes the structure of objects of that *type,* or *class,* together with the set of operations appropriate to those objects. Objects can be defined as extensions of existing ones by a mechanism called *inheritance.* The specification 'B inherits A' in the definition of object B means that B contains the data and operations defined for A in addition to those specifically defined for B. Inheritance may be viewed as an abbreviation mechanism that avoids redefining the attributes of an already introduced object in the definition of another.

One advantage of using object-based programming languages is that they facilitate the creation of software modules that closely match the problem domain, an important feature for building understandable programs. Conventional languages, such as Pascal, often lead to program structures radically different from the structure of the problem domain. The reason is that in such languages there are two kinds of entities: data items, which are passive and represent the information of the program, and procedures, which manipulate the data. The programmer in a conventional language can either map the problem domain into a set of procedures, or can map the problem domain to the data, and then define procedures that transform the input data to the output data. By contrast, object-based programming allows the programmer to concentrate on the abstractions to be manipulated - procedures and data can be treated as indivisible aspects of objects in the problem domain. Many programs can be designed by straightforwardly identifying the objects in the problem domain, and deciding how to implement the objects' behaviour.

The term architecture, mentioned above, is used in a broader meaning than just hardware organization. Today's *architecture* must include software as much as hardware, since developments on both fronts may be necessary to achieve the most useful and cost-effective systems for the end users.

1.2. Distributed architectures

An architecture may be termed *distributed* when the discrete elements of the overall processing activity may be located in more than one component, at more than one geographical location. The components forming a distributed architecture do not share primary memory, and so communication and coordination via shared memory techniques is generally not applicable. Instead, message-passing in one form or another is employed. This thesis will focus on distributed architectures composed of a number of autonomous workstations or personal computers (nodes) communicating via a local area network. A node in such a network will typically contain various processes providing services, for example data retention, that can be used by local and remote processes.

In the following, the major advantages of distributed architectures in the areas of reliability, security and performance are discussed. In a distributed architecture, individual nodes are physically independent from each other. Regions affected by a failure have well-defined physical boundaries - a hardware failure in one node usually has no direct impact on others. This feature makes recovery and reconfiguration possible. In addition, distribution allows security to be based on the existence of physical domains between which communication can be strictly controlled, rather than on logical barriers. Distributed architectures can also offer increased power through parallel processing, provided one has decomposed the overall task into parallel subtasks with minimum communication requirements.

Despite all these advantages, however, a number of difficult issues are still latent. Many of these are related to component (node) failures and to slow communications, and have important implications for the distributed system's reliability. While the increased number of interconnected nodes may remove the single point of failure of a centralized architecture, computations will become susceptible to remote node crashes and communication failures. Therefore any well-engineered distributed system should strive to cope with situations where the system is partly running and partly crashed. Dependencies between individual components should be minimized and distributed error detection and recovery should be favoured by providing each component with local mechanisms without relying on the well-functioning of the rest of the architecture. Moreover, since communication between components of a distributed architecture are typically orders of magnitude slower than in a centralized one, distribution requires the optimization of communication between nodes, and the grouping together of applications which need high communication rates.

Many research projects have employed the object-based methodology for the design of reliable distributed architectures, such as Eden [Almes85], Argus [Liskov87], and Arjuna [Shrivastava88a]. It appears that this methodology offers significant advantages for the design and implementation of such architectures. For example, the *modularity* afforded by an object-based architecture simplifies the implementation of recovery from failure, and reconfiguration; and the inheritance mechanism can provide a controlled means of introducing recovery within objects, as shown in [Dixon87, Shrivastava88a]. A further advantage of the object-based methodology is *protection*. This facility is usually provided to constrain the way information is used and changed. In object-based architectures, physical domains (nodes) can be decomposed further into logical domains (objects). The only way the user can act upon an object is by operation invocation, so a straightforward technique for constraining arbitrary manipulation of an object is to constrain the ability to perform operations on that object.

However, distributed object-based architectures are affected by a number of new problems related in particular to storage management. In object-based architectures, a new object is allocated explicitly, but there is no explicit construct that causes an object to be deallocated. Such a construct would be unsafe, because it could be employed to deallocate an object even though the object was still in use. The use of references to deallocated objects would be likely to cause inconsistent behaviour. Because there is no explicit deallocation in many object-based architectures, the storage manager must identify objects that have become *inaccessible* and deallocate them automatically. This task is known as *garbage collection*. Various problems arise in distributed object-based architectures relate to garbage collection. One problem is that it is impractical, if not impossible, to stop the entire system while collecting unused objects. Another problem that occurs in these architectures is that objects may be retained for a long period of time. That means the total number of objects may grow extremely large in relation to the processing power available to collect unused storage. An incremental garbage collection scheme, which allows some garbage to be collected without examining the entire system storage, is essential when the time required to access every object in the system would be prohibitive. Once again the issue of reliability becomes important. The collection scheme must be able to tolerate component failures. It should be possible to continue the collection after a failure without incorrectly deallocating or reallocating storage for objects in use, and all unused objects should be ultimately collected.

This thesis explores the problems to be solved for achieving reliable object-based computing in the face of node and communication failures, and also presents a garbage collection scheme that is suitable for use in a distributed unreliable system.

1.3. Structure and aims of the thesis

This thesis concentrates on the provision of support for one particular property of distributed systems, the property of reliability. Most projects which are addressing this area of research have concentrated on producing new languages or operating systems that provide the necessary support, and much research effort has been concerned with the use of object-based methodology. It is not the aim of this thesis to design a new programming language or operating system, but rather to put the current research effort into a new perspective, which shows that work on object-based architectures can also have relevance to more conventional architectures.

The thesis is organized as follows. Chapter 2 expands on the background to the work described in this thesis. In order to discuss the similarities and differences among the variety of approaches to object-based programming, the terminology employed in this thesis will be introduced in Chapter 2 and related to other work. The most relevant features of object-based programming will be characterized, and various programming methodologies worthy of special study will be identified.

In Chapter 3, two techniques will be presented which can be employed to introduce object-based features, such as the sub-classing form of inheritance provided by Smalltalk and Simula, into a procedure-based language. Although the object-based and procedure-based approaches are apparently dissimilar, a family of procedure-based languages will be described which allow the programmer to profit from object-based ideas.

Chapter 4 discusses reliability issues concerning the design of distributed architectures. In particular, this chapter concentrates on the main causes of unreliability, illustrating these with some general solutions and examples. Among the issues considered are communication failures and node crashes.

Chapter 5 examines the structure of distributed architectures incorporating error recovery, and proposes their partitioning into two broad categories. Two canonical models, each representing a particular category of architectures will be constructed. The first model, called object-based, incorporates objects as the entities for program construction while the second model, called process-based, employs communicating processes. Arguments and examples will be presented to show that the object-based model and the process-based model are the dual of each other. As a consequence of the duality, techniques and mechanisms which have been developed within the domain of just one of the models can be mapped and applied to the other model. This point will be illustrated by mapping some well-known object replication techniques developed within the context of the object-based model to the process-based model thereby revealing some interesting process replication techniques.

The techniques described in Chapter 3 also allow the creation of objects at run-time and require, as most object-based architectures do, an automatic garbage collection facility for storage management. A distributed garbage collection facility will be needed for such architectures, if they permit access to remote objects. Chapter 6 describes in detail the design and implementation of a novel garbage collection scheme for distributed architectures. The proposed scheme achieves its task despite the occurrence of commonly encountered failures in distributed systems (such as lost messages and node crashes), performs in parallel with the other system activities, and is capable of dealing with both volatile and stable objects.

Chapter 7 concludes this thesis by reviewing its objectives, providing some concluding remarks, and discussing possible future developments of the work that has been presented.

Chapter 2

Object-Based and Object-Oriented Programming

Despite the fact that a large computing community is working with object-based and object-oriented programming systems and languages, there is still a fair amount of confusion over what the terms mean and what terminology to use. To quote the report of the discussion sessions of the European Workshop on Object-Oriented Programming ([Wegner88], p. 21):

> *"The discussion clearly demonstrated differences of perspective and exposed a lack of precision in the definition and use of some fundamental concepts in object-oriented programming."*

Therefore, it is appropriate to give here the present definitions, and to discuss the similarities and differences among the variety of approaches to object-based programming.

A tremendous amount has been written about object-based programming, some of the better surveys being [Cardelli85, Wegner87a, Stroustrup88]. The aim of this chapter is to avoid going over too much of the material that is readily available, but to provide a somewhat different survey. The survey that follows can be contrasted with those mentioned above by virtue of the much simpler classification that it uses; specifically the classification presented brings out the most important issues and concentrates on just three topics: encapsulation, abstraction and inheritance.

2.1. Definitions and examples

The following properties are considered to characterize the relevant features of object-based programming: *encapsulation, abstraction,* and *inheritance.* Encapsulation is the strict enforcement of the principle of *information hiding* advocated by Parnas [Parnas72]. Encapsulation allows software components to be implemented and reimplemented independently, and is important for supporting modifiability and reliability of software architectures by controlling and constraining the way software components can interact. At the programming language level, encapsulation can be provided by means of a *data abstraction* mechanism [Hoare72, Liskov77]. Besides encapsulating data and operations into abstract data types, it is further possible to organize abstractions into a hierarchy [Dahl72]. This hierarchy serves to relate similar abstractions by an inheritance relationship [Snyder86a]. Inheritance allows an abstraction to inherit various (or all) characteristics from another abstraction higher in the hierarchy.

In the following subsections, the above features of the object-based architectures will be discussed, and the terminology used in the rest of this thesis will be introduced.

2.1.1. Objects and classes

Following a standard terminology [Jones78], an *object* is an entity out of which a structural model of a system is built. The important feature of an object is that no other object within the same system has any means of finding out what is 'inside'. The key concept here is encapsulation. Generally, an object encapsulates some state together with the set of operations that are the only means by which that state can be manipulated. The result of invoking an operation of an object depends on the object state as well as on the operation arguments.

A system will often contain many similar objects. For example, a window management system may have several windows which, with the exception of their location and size, exhibit identical behaviour. Dahl, Dijkstra, and Hoare asserted in their book ([Dahl72], p. 177):

> *"Any useful concept has some degree of generality, i.e. it is a class of specialized instances. In other words one tries to group phenomena occurring in a dynamic system into classes of phenomena and describe each class by a single piece of program."*

At the programming language level, the implementation of similar objects can be collectively defined by declaring a *class*. A class is a description of the common features of similar objects from which an individual object may be created. This notion of class applies to what is termed *cluster* in CLU [Liskov81], *type* in Trellis/Owl [Schaffert86], and *class* in Smalltalk [Goldberg83], Simula [Dahl70], and C++ [Stroustrup86]. A class characterizes the behaviour of its objects by defining the only operations that can manipulate the state of its objects.

An operation has access to several kinds of variables which contain the object state, but which differ in terms of how widely they are available and how long they persist. These various kinds of variables can be divided in (1) *instance variables,* which are private variables accessible only to a single object, and (2) *class variables,* which are shared by all the objects of a single class.

Although the terminology just introduced is among the most popular in the current literature on object-based programming, there are some well-known systems that employ different terms for the same basic concepts. In C++ [Stroustrup86], for example, the operations that can manipulate the objects of a class are termed *public member functions,* the instance variables are termed *private member variables,* and the class variables are termed *static member variables.* In Smalltalk, objects are also known as *instances* of their classes, operations defined on objects are called *methods,* and objects are manipulated by applying methods to them. The only way to apply a method to an object in Smalltalk is to send a *message* containing the method name and the parameters to that object. The object responds to the message by possibly changing its state and by returning a result object. It is worthwhile to note that although the concept of message passing appears to be radically different from the conventional concept of procedure call, the difference is more pedagogical than semantic. Message passing emphasizes the caller's lack of knowledge of the code body which will be executed. However, any procedure call can be viewed as a message send, and vice versa. Examples of object-based programming languages whose authors describe them using notions of procedure-calling rather than message-passing are: Simula-67 [Dahl70], Trellis/Owl [Schaffert86], and C++ [Stroustrup86].

To illustrate the use of objects and classes, consider an outline of a program for displaying rectangular regions on a display screen. Figure 2.1 gives a class definition for the class Box, using a syntax similar to that of Smalltalk, though simpler. The first four instance variables store the coordinate and size information of a box, and the last instance variable records the shade which fills that box. The origin of a box in the coordinate system is determined by the instance variables xOrigin and yOrigin (the use of upper-case letters in the middle of a word is part of the established Smalltalk style), and the default origin of objects of this class is defined as (100, 200). The size of a box is determined by instance variables xLength and yLength and the default size is 10 x 30. Operations on a box include moving it to a new origin, changing its size, displaying it, and changing the shading inside the box. The bodies of the operations, which for the sake of brevity have been omitted, are assumed to follow the operation headers. Messages define an interface for interacting with boxes. Some examples of such an interaction are also shown at the bottom of Figure 2.1. Syntactically a message is composed out of an object name, followed by a *:selector* indicating the required operation, followed by any further arguments, and terminated by a period. For example, objects of the class Box are created by sending Box a *new* message, and an object can be moved by sending it a *move* message. Users can only manipulate a box by the relevant operations - they do not need to know the implementation of a box in terms of the

```
class Box
class operations
    new;
instance variables
    xLength 10;
    yLength 30;
    xOrigin 100;
    yOrigin 200;
    defaultShade white;
instance operations
        % changes origin in the display %
    move: newXOrigin, newYOrigin;
        % changes the location and axes of the box %
    reshape: newXOrigin, newYOrigin, newXLength, newYlength;
        % fills the inside of the box with a new shade %
    shade: newShade;
        % displays the box %
    draw
        % move a box at the outmost level in the screen %
    top: originX, originY, lengthX, lengthY;
end class

Objects of class Box can be created and manipulated as follows.

b1 ← Box :new.
b2 ← Box :new.
b1 :move 15 25.
b2 :reshape 10 10 20 20.
```

Figure 2.1: Example of class and objects.

instance variables and their manipulations.

2.1.2. Sub-classes and inheritance

So far it has been seen that each object belongs to exactly one class. In some cases, it would be convenient if objects of one class could also be used as objects of another class. Such sharing of objects among classes can be achieved by employing a sub-classing mechanism. One class may be a *sub-class* of another (its *super-class*), with the implication that if B is a sub-class of A, an object of class B may be used wherever an object of class A can. In other words, objects of class B can also be seen as object of class A.

Connected with sub-classes is the concept of *inheritance.* A class may share or *inherit* various characteristics of its super-class, and may have its operations inherited by sub-classes. The inherited characteristics may include the operations and instance variables of the super-class, and these characteristics may be extended or restricted in the sub-class. If an operation is to be executed on an object, the search for the operation definition begins in the class of the object and if unsuccessful there proceeds to the super-class of that class, and so on. The first definition of the operation that is found is executed. Hence, an operation defined in a sub-class hides an operation of the same name in the super-class. This hiding mechanism allows a sub-class to customize the more general characteristics of its super-class. The sub-class in turn may pass on its own or inherited characteristics to its sub-classes. It can be seen that this is in the tradition introduced in Algol-60 for the scope of local and non-local identifiers [Randell64].

The inheritance structure discussed above is strictly hierarchical since it allows a class to have one super-class only, and specifies that objects of a sub-class can be used as objects of the super-class. This approach was first followed in Simula-67, and was adopted in the first version of Smalltalk-80.

As an example of sub-class in the window manager system, suppose that one wants to define windows that display text. These new windows would be objects of a sub-class of the ordinary class Box. The new class, called TextBox in Figure 2.2, adds a new instance variable to keep the font of the text, and redefines the operations inherited from Box.

```
class TextBox
superclass Box
instance variables
    font     roman;
instance operations
    - - -
        % display a TextBox %
    draw
        super :draw;
        printText;
private operations
        % display the text in the proper font %
    printText;
end class
```

Figure 2.2: Example of sub-class.

For example, the *draw* operation of TextBox first draws a box, and then uses the font information to display the text on the screen.

One problem with almost all inheritance mechanisms is that they compromise encapsulation to an extent. An inheritance mechanism establishes a second sort of user of a class C - the *inheriting* users, namely C's sub-classes, alongside the *instantiating* users, who create objects of the class and use the C's objects by calling the operations only. While the instantiating users do not see the representation of the objects they manipulate, C's sub-classes are typically permitted to violate encapsulation. For example, Smalltalk makes every characteristic of class C public to its sub-classes - the code in the operation of a class may directly access even those instance variables that were defined in the super-class. On the one hand, permitting access to instance variables defined by super-classes can compromise the benefits of encapsulation - a change to a class can affect all its sub-classes. On the other hand, to be able to take full advantage of the sharing of the implementation code, it can be argued that a sub-class programmer should have the same privileges as the person who originally wrote the super-class code, and hence should be allowed access to the internal structure of the super-class.

Some architectures have recognized that an alternative to the Smalltalk approach is required, and have provided separate interfaces to inheriting and instantiating users - sub-classes can access the super-class through a well-defined interface, but they may employ operations not available to instantiating users. For example, in Trellis/Owl, a characteristic of a class declared to be *subtype-visible* is visible to all its sub-classes in the inheritance hierarchy but invisible to instantiating users. A comprehensive discussion of this issue is given in [Snyder86a], where it is suggested that instance variables should be protected from direct access by requiring the use of explicit access operations, as in Trellis/Owl.

A generalization of the single inheritance hierarchy, where a class can have only a single super-class, is to allow a class to have multiple super-classes.

This feature is called *multiple inheritance*. Multiple inheritance allows a given class to inherit characteristics from one or more classes. Trellis/Owl and Traits [Curry84] are examples of programming languages incorporating a class hierarchy with multiple inheritance. Multiple inheritance appears to be important in situations where a class can be created out of a combination of independent super-classes. In the window management example, the programmer could define the class TextBox as inheriting both from the class Box and from the class InputOutput, as shown in the lattice of Figure 2.3.

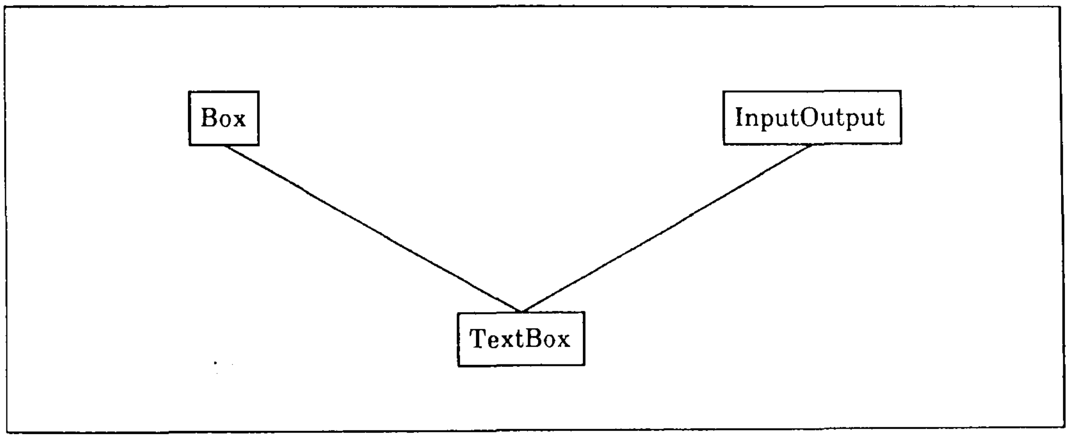

Figure 2.3: Example of multiple inheritance.

In this way, the implementation of the class TextBox can be further simplified. For example, the implementation of the *printText* operation of class TextBox can reuse the relevant operations of class InputOutput.

As yet, there is no agreement about the semantics of multiple inheritance, and various multiple inheritance schemes are being examined and weighed against the complications they add to the implementation of an architecture. One basic issue concerns the rules for resolving the conflict among the super-classes, where instance variables or operation names inherited from more than one super-class 'collide', that is have the same name. The way this conflict is resolved in most object-based architectures leads to a violation of encapsulation. For example, in Extended Smalltalk [Borning82], a version of Smalltalk-80 with multiple inheritance, it is an error if a collision occurs. This compromises encapsulation since a change in a class could cause sub-classes to become illegal. For example, considering the inheritance lattice in Figure 2.3, if an instance variable in class Box is renamed, it may cause sub-class TextBox to become illegal if class TextBox also inherits from class InputOutput an instance variable with the same name as the renamed variable in Box. In Trellis/Owl, the programmer of the sub-class must resolve explicitly such a conflict by specifying which variables he wants to inherit. This leads to similar problems with encapsulation as in Extended Smalltalk, since renaming instance variables of a class may be visible to the programmer of the sub-class, that is he may be required to resolve new collisions that may arise.

Another basic issue with multiple inheritance concerns multiple occurrences of a super-class along different inheritance paths. Several possibilities of handling such a situation have been proposed. For example, suppose a class A inherits from classes B and C which both inherit from a class D: in the approach adopted by C++ and Trellis/Owl the class D is inherited just once; in a second approach adopted by CommonObjects [Snyder86b] and ThingLab [Borning81] a copy of D is inherited along each path; in a third approach adopted by Flavors [Moon86] and CLOS [Bobrow87] the classes A,B,C, and D are interposed so that the hierarchy is linearized and the class D is inherited once.

This section has presented the main features and relevant issues of object-based architectures, and has discussed similarities and differences among the most popular approaches to object-based programming. The next section attempts a classification by grouping together similar object-based architectures and by relating the relevant features of each group.

2.2. A classification

In this section, object-based architectures are characterized in terms of the notions of object, class and inheritance. Various families worthy of special study are identified. Figure 2.4 shows some of the most popular members for each of the families considered.

In this thesis the term *encapsulation-based architecture* is used to mean a software architecture which provides facilities that makes it possible (reasonably easy, safe, and efficient) to use encapsulation. The encapsulation-based architectures include all architectures which support objects, like Ada [DoD80], Actors [Agha86], Modula 2 [Wirth83], and PS-Algol [Atkinson87]. An encapsulation-based architecture is termed here *class-based* if each object has a class which characterizes its behaviour; and an encapsulation-based architecture is *inheritance-based* if it supports objects and inheritance. Previously, inheritance has been defined as a mechanism for sharing

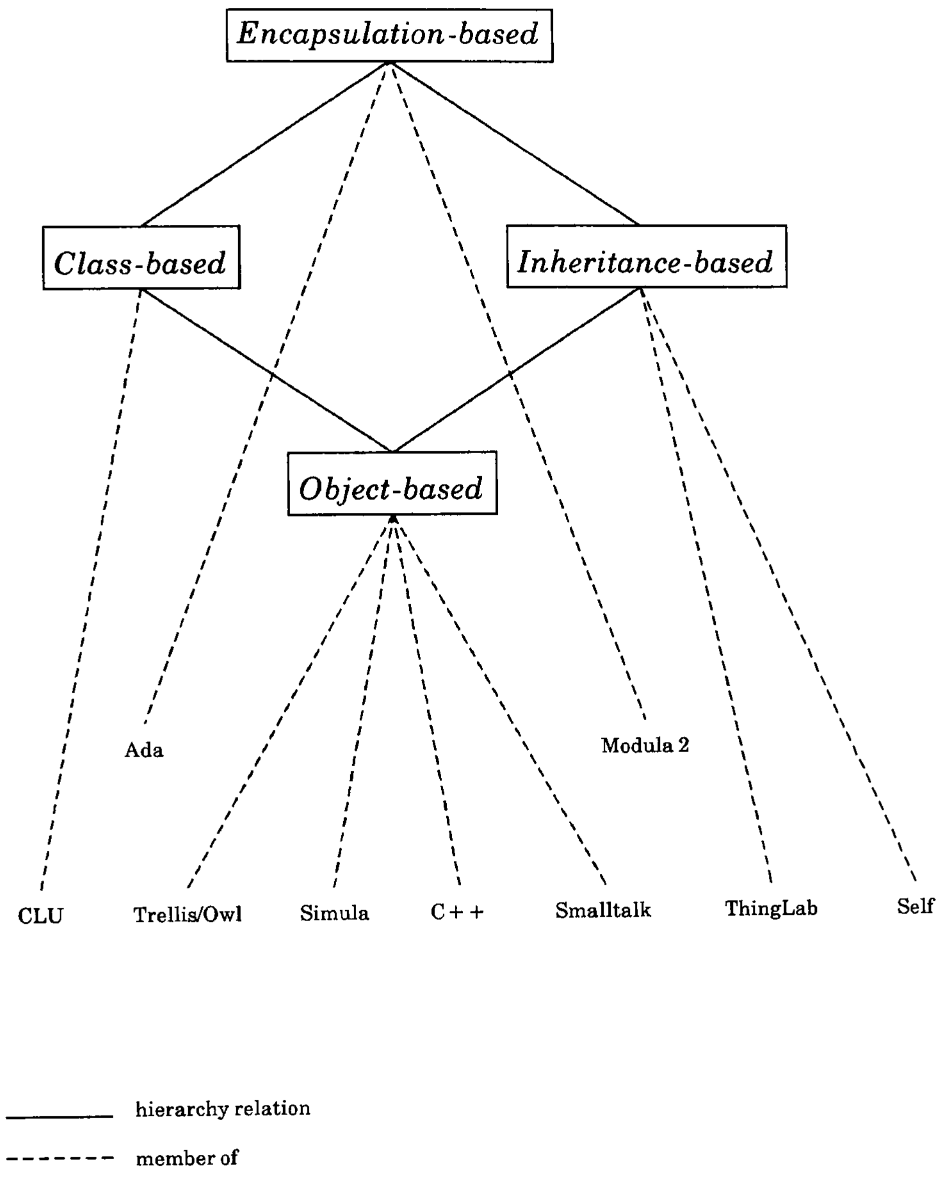

Figure 2.4: Lattice of object-based programming.

characteristics in class hierarchies. However, there are various architectures which provide a variety of other forms for sharing characteristics. For example, Self [Ungar87] and ThingLab [Borning81] do not include classes; instead every object is regarded as a *prototype* for object creation [Borning86], and a form of inheritance among objects is provided. In the literature, the term inheritance is sometimes loosely employed to denote a variety of sharing techniques [Cook88], and sometimes the term *delegation* is used [Lieberman86, Stein87]. In the context of inheritance-based architectures, the term inheritance is used here to mean a more general class-independent term for sharing, which allows one implementation of an object to be related to another hierarchically.

In this thesis, an encapsulation-based architecture is termed *object-based* if its objects belong to classes, and the classes themselves can be related by an inheritance mechanism and organized into a class hierarchy. According to the above terminology the family of object-based architectures is smaller than the family of encapsulation-based architectures, since object-based architectures exclude architectures like Ada, and CLU but include architectures like Smalltalk, Trellis/Owl, and Simula.

As shown in Figure 2.4, class-based architectures such as CLU, and inheritance-based architectures such as Self and ThingLab, are also encapsulation-based architectures. Ada is an example of an encapsulation-based architecture that is neither class-based nor inheritance-based. Ada does not fully support an inheritance mechanism. Although it does provide subtypes and derived types, these are only means of restricting a general type, to some specific range or purpose. Ada does not make it possible to extend a pre-existing type by adding additional variables and operations. Moreover, although Ada provides an encapsulation mechanism by allowing the definition of a *package,* Ada packages are just program units and cannot be considered classes, as argued in [Wegner83]. On the other hand, CLU is a class-based architecture, which allows a perfect match between the syntactic concept of a cluster, and the semantic concept of a class. However, CLU does not have an inheritance mechanism for defining hierarchical relations between classes, and is therefore an example of a class-based architecture that is not object-based.

Incidentally, it should be noted that Figure 2.4 may also be seen as representing an example of an object-based hierarchy. In the lattice shown, extant architectures may be regarded as objects (e.g. Ada, Simula, etc.), and programming methodologies that group together architectures with common features may be thought of as classes (e.g. Encapsulation-based, Object-based, etc.). An inheritance mechanism allows the organization of the four programming methodologies into a hierarchy. For example, class *Class-based* may be defined as a sub-class of class *Encapsulation-based,* since *Class-based* can inherit the definition of object from *Encapsulation-based.* Moreover, class *Object-based* illustrates an example of multiple inheritance - *Object-based* inherits the definition of class from *Class-based,* of inheritance from *Inheritance-based,* and of object from *Encapsulation-based.*

In the literature is also common to encounter the term *object-oriented* in addition and/or in contrast to the term 'object-based'. Sometimes, the term 'object-oriented' is considered equivalent to 'object-based', as in [Stroustrup88, Liskov88]. Sometimes, programming languages such as Ada or Modula-2 are considered to be 'object-based' even though they do not possess all the properties of objects in Smalltalk or Simula, and the term 'object-oriented' is thought of as stronger than 'object-based', and is used to denote programming languages with additional features, as for example in [Hendler86, Wegner87b].

In this thesis, the architectures that provide the linguistic features of object, class, and inheritance are called 'object-based'. However, when considering the implementation features of object-based architectures, it could be sometimes convenient to make a further distinction. Let us consider, for example, the object-based extension of the C programming language [Kernighan78], that is C++. The current implementation of C++ cannot be regarded as object-based, because some of the underlying C features, for example the standard types (integer, etc.), which C++ takes in as part of its implementation, are not object-based. So if one has those features in an architecture together with the object-based constructs then the architecture as a whole cannot be considered truly object-based. Of course, one could implement in C++ a library of classes which redefines all the standard C types according to the object-based methodology. It is thus useful to distinguish C++ from other architectures where everything is implemented as objects. In this respect, we feel that C++ may be regarded as object-oriented, because is 'orienting' the programmer in the right direction, while architectures like Smalltalk may be regarded as object-based, that is the term 'object-oriented' may be thought of as weaker than 'object-based'. This distinction, though useful, is out of the scope of the above classification, and will not be explored further.

2.3. Concluding remarks

A characterization of the relevant features of the object-based programming methodology has been discussed. Based on their dependence relations four approaches have been identified. These approaches include the object-based methodology where objects, that is encapsulation of data and operations, are grouped into classes through which the concept of abstract data types is provided. The classes themselves can be organized into a class hierarchy. Such hierarchies allow similar classes to be related together in such a way that the code implementing the behaviour of one class can be automatically re-used (inherited) by classes lower in the hierarchy, which simplifies the implementation of those lower-level classes.

Throughout this chapter, we have been talking about object-based programming as though it can only be done with special programming languages. In fact, one of the avenues that has been explored in the present research has been the use of object-based programming techniques in a procedure-based language. This work, that will be described in the next chapter, arose from particular work on distributed systems and in particular garbage collection in distributed systems. These two later topics are deferred until subsequent chapters.

Chapter 3

Object-Based versus Procedure-Based Programming Languages

This chapter addresses the possibilities of exploiting procedure-based languages to allow an object-based style of programming. Although the two approaches are apparently dissimilar, a family of procedure-based languages is considered which allows the development of techniques which enable an efficient implementation of object-based features.

The techniques provided allow one to obtain the benefits of object-based programming, such as the sub-classing form of inheritance provided by Smalltalk and Simula, while preserving the static binding and strong type-checking features of the language. In particular, it will be discussed how class can be defined in procedure-based languages, how sub-classes can specialize their super-classes and how the association of operation names and routines is affected.

The arguments devised can also be seen as addressing the issue of whether and to what extent procedure-based languages limit the ability of programmers to adopt an object-based style of programming with a class hierarchy structure.

3.1. Motivations

The techniques, whose design and implementation are described in this chapter, are intended to provide an adequate support for programming large systems. The various motivations that led to the development of these techniques can be traced back to the author's earlier experiences in the design and implementation of distributed object-based systems.

In early work on object-based systems, the author was involved in the implementation of the Fault-tolerant Distributed Garbage Collection (FDGC) mechanism described in [Mancini87] for the Flex system. The Flex system [Foster82] is a computer architecture which has been developed at the Royal Signals and Radar Establishment (RSRE) of the U.K. Ministry of Defence and is available in the Computing Laboratory on the ICL Perq 2.

The development of the FDGC was completed successfully, employing the RS Algol-68 compiler [Woodward83] provided by the Flex system - a compiler which supports a version of Algol-68 extended with first-class higher-order procedures. However, during the implementation, it became clear that the use of object-based programming techniques would be advantageous and a way was found of achieving, through disciplined use of RS Algol-68, what was in effect object-based programming in a non object-based programming language.

In the light of this experience, the author set out to investigate the design and implementation of techniques to enable an efficient implementation of object-based features. Since Flex has been used in the Computing

Laboratory for experiments in distributed object-based computing, the implementations were studied in the context of the Flex system.

Thus, the techniques reported in this chapter were developed, one of which has also been described in [Mancini88a]. These techniques proved to be an improvement on standard Algol-68, and a useful tool for constructing large programs. In particular, they provided a convenient base for the colleagues at RSRE to develop experimental versions of a new graphical user interface for the Flex system. The RS Algol-68 compiler supported by the Flex system has been used to experiment these ideas, although the techniques proposed are applicable to every procedure-based language. What follows illustrates them using the ML language [Milner84], because of the wider knowledge that exists of this language than RS Algol-68.

This chapter is organized as follows. Section 3.2 examines different ways of providing data abstraction and encapsulation. Section 3.3 discusses various issues about providing sub-classing in a procedure-based language. In Section 3.4 a first technique for sub-classing is presented in detail - it exploits polymorphic procedures and requires some run-time lookup. In Section 3.5 a second technique is presented - all bindings are established at the time objects are created and run-time lookups are avoided. Performance considerations and conclusions from this study are discussed in the final sections.

3.2. Class definition

This section explores how classes can be defined in procedure-based languages so as to provide encapsulation and data abstraction. A procedure-based language with the following characteristics will be employed to express the techniques proposed: *polymorphism, first-class higher-order procedures, static binding,* and *strong type-checking.*

There are several kinds of polymorphism investigated in the literature [Cardelli85]. In this chapter a language is considered to be *polymorphic* if some of its expressions and variables may have more than one type, and a procedure can work uniformly on a range of types; these types normally exhibit some common structure. Polymorphism is desirable in modern programming languages because it enables the writing of extremely general-purpose programs in a transparent manner - the bare algorithm and no superfluity, as argued in [Harland84].

Traditionally, procedures have been included in programming languages as denotations, not as proper values. By *first-class higher-order* procedures [Abelson85], it is meant procedures that can be treated just as any other value in the language. First-class higher-order procedures can be accepted as arguments to other procedures, stored in variables, and returned in the results of a procedure. As it will be shown in the following, first-class higher-order procedures provide an alternative mechanism for implementing encapsulation to the selective exporting of local names, let alone an increased uniformity in the language.

In the language considered, variables are bound *statically,* that is, a free variable in a procedure gets its value from the environment in which the procedure is defined. This means that the binding of a variable in a program is determined by the static structure of the program, not by its run time behaviour. The language is further assumed to be *strongly typed,* that is it is guaranteed that programs will execute without type errors - the type compatibility of all expressions and variables can be determined from the static program representation at compile-time. While the advantages of the static binding are in terms of higher efficiency by comparison with solutions employing dynamic binding, the strong type-checking helps to recognize sources of error early and therefore increases the degrees of correctness, testability, and maintainability of programs.

In some object-based languages (for instance, Smalltalk) a class is itself an object, and creating objects of the abstract data type represented by that class involves applying a create operation (new in Smalltalk) to that class object. A similar way of providing encapsulation is by means of procedural abstractions. As has been pointed out in [Horning76], the advantages and aims of procedural and data abstraction are similar. Just as a procedure separates the implementation of a function from its use, so the abstract data type separates the representation of an object from its use. Indeed if procedures are first-class entities, the mechanism for both abstractions can be the same - that of procedures. In languages providing first-class higher-order procedures, a class description can be given by declaring a procedure which, when executed, creates an object, and returns the set of procedures (operations) which can be applied to that object. (A detailed presentation of techniques for providing data abstraction and encapsulation through the use of higher-order procedures can be found in [Abelson85], although techniques for handling sub-classes and inheritance are not covered.)

In what follows, the example of the window management system of Chapter 2 will be reconsidered, in order to illustrate the similarities between the object-based and procedure-based approaches. The same encapsulation features of the object-based languages can be achieved in procedure-based languages by suitable use of first class higher-order procedures. For example, the dual of the program for the class Box in Figure 2.1 is shown by the ML program of Figure 3.1 - the procedure *makeBox* creates an object of the class Box hiding the five instance variables behind the procedural interface composed of the five operations to manipulate them. Suitable functions must be defined in order to select and invoke one of the five operations of the tuple returned by *makeBox*. It is worthwhile to note that such selector functions are polymorphic, for example, *moveOF* in Figure 3.1 can return a value of different types at different times depending on the type of the first component of the tuple passed as actual parameter.

This simple example shows that the encapsulation features of both approaches are similar. In particular, it could be pointed out that:

- In both types of languages, objects may be created dynamically at run-time. This can be achieved in the object-based approach by sending a message *new* to the class Box, and in the procedure-based language by

```
% creates an object of type Box and returns the operations to manipulate it %
val makeBox = fun () ⇒
    let
        % instance variables %
        val xLength = ref 10 ;
        val yLength = ref 30 ;
        val xOrigin = ref 100 ;
        val yOrigin = ref 200 ;
        val defaultShade = ref white ;
    in
        % returned instance operations%
        (   fun newXOrigin, newYOrigin: int ⇒ ... ,
            fun newXOrigin, newYOrigin, newXLength, newYlength: int ⇒ ... ,
            fun newShade: shading ⇒ ... ,
            fun () ⇒ ... ,
            fun originX, originY, lengthX, lengthY: int ⇒ ...   )
    end ;
```

Given the following declarations:

```
val moveOF = fun (move, reshape, shade, draw, top) ⇒ move ;
val reshapeOF = fun (move, reshape, shade, draw, top) ⇒ reshape ;
...
```

Boxes can be created and manipulated as follows.

```
val b1 = makeBox ;
val b2 = makeBox ;
moveOF b1 (15, 25) ;
reshapeOF b2 (10, 10, 20, 20) ;

...
topOF b2 (10, 10, 20, 20) ;
```

Figure 3.1: Definition of Box written in ML using first-class higher-order procedures.

calling the procedure *makeBox*.

- This programming discipline for procedure-based languages allows the use of procedures employing a style of programming equivalent to that of the object-based approach. Instead of passing objects to operation routines, objects are required to perform operations on themselves. The code of the routines can be factored out into the procedure which represents the class. For example, the code of procedure *makeBox* contains the routines *move, reshape, shade, draw,* and *top* which are referred to by the class objects *b1* and *b2* in order to manipulate their own instance variables.

- Many object-based languages, such as Smalltalk, perform run-time type checking, while in the case of procedure-based language compile-time checking is possible.

3.3. Sub-classes and inheritance

As mentioned in Chapter 2, another common property of the object-based approach is that it allows a set of classes to be organized as a class hierarchy. The class structure as described so far does not permit one class to inherit characteristics from another class.

To clarify this idea, reconsider the example of the class Box of of Figure 3.1. Suppose that another kind of box, BorderedBox, is required with a visible border that frames it on the display. Objects of class BorderedBox would be essentially an object of Box but with a border. This overlap of characteristics suggests that it would be desirable to be able to exploit inheritance to obtain the main characteristics of BorderedBox by specializing the class definition of Box.

The example in Figure 3.2, where the keyword class has been substituted for proc for clarity, shows the class BorderedBox defined as a sub-class of the class Box. Class Box contains basically the same operation definitions as in the program of Figure 3.1. In class BorderedBox, which inherits operations from class Box through the superclass declaration, four operations are added together with a new instance variable for recording the border size. The first operation of BorderedBox, *reshape*, is a specialization of the operation of the class Box. This specialization is required because the border needs to be redrawn when a box is increased in size. The *reshape* operation of BorderedBox also needs to use the *reshape* operation of its super-class, and achieves this behaviour through the use of the pseudo-variable *super*. The specialized *reshape* of BorderedBox also invokes the operations *draw* and *erase*, local to BorderedBox, to draw and erase the border. These operations are for internal use, while the *setBorder* operation is part of the external interface.

Procedure *top* in class Box also illustrates an additional requirement. It is assumed that *top* requires a *reshape* operation to be executed, and that class BorderedBox inherits the operation *top* from the super-class Box. The question arises as to how *top* can invoke *reshape*, differentiating between that defined in class Box and that in class BorderedBox. This is where the use of the pseudo-variable *self* comes in. By referring to 'reshape of self', the

```
class Box;
begin
    instance variables
    int xLength  = 10;
    int yLength  = 30;
    int xOrigin  = 100;
    int yOrigin  = 200;
    shading defaultShade = white;

    instance operations
    proc move (int newXOrigin, newYOrigin) ... ;
    proc reshape (int newXOrigin, newYOrigin, newXLength, newYlength) ... ;
    proc shade (shading newShade) ... ;
    proc draw ... ;
    proc top (int originX, originY, lengthX, lengthY):
    begin
        - - -
        reshape of self (m, n, p, q);
        - - -
    end
end Box;

class BorderedBox;
superclass Box;
begin
    instance variable
    int borderSize = 2;      % width of the border %

    instance operations
    proc reshape (int newXOrigin, newYOrigin, newXLength, newYlength):
    begin
        eraseBorder;   % erase old border %
            % now reshape box as before %
        reshape of super (newXOrigin, newYOrigin, newXLength, newYlength);
        drawBorder     % draw new border %
    end;
    proc setBorder (int newBorderSize) ... ;      %set a new border size%

    private operations
    proc eraseBorder ... ;
    proc drawBorder ... ;
end BorderedBox;
```

Bordered boxes can be created and manipulated as follows.

```
BorderedBox bb1 := new;
BorderedBox bb2 := new;
setBorder of bb1 (3);
move of bb1 (15, 25);
reshape of bb2 (10, 10, 20, 20);
top of bb2 (10, 10, 20, 20);
```

Figure 3.2: Example of class hierarchy.

search operation for the *reshape* operation begins in the class of the object on which the *top* operation was originally invoked, not in the class in which the code for *top* is declared.

At the end of Figure 3.2, a possible use of these class descriptions is shown. Two objects, *bb1* and *bb2*, of the class BorderedBox are created. The border of *bb1* is first enlarged, by 'setBorder of bb1(3)', and then is moved, by 'move of bb1(15, 25)'. Similarly, object *bb2* is first reshaped using the specialized *reshape*, and then is brought to the top. It is worth noting that the last operation requires the invocation of *top* defined in the class Box, and that operation *reshape* called in the body of *top* has to be bound to the procedure defined in class BorderedBox regardless of the fact that a reshape procedure is present in Box - the *reshape* defined in Box cannot handle BorderedBox.

In the next sections, two possible implementation of the sub-classing mechanism in a procedure-based language will be shown. The issues that will be discussed include:

Compile time errors.

For example, because of the strong type-checking feature of the language the compiler will complain about the instruction 'move of bb1(15, 25)' in Figure 3.2 - *move* is not declared among the procedures of the class BorderedBox. Moreover, a treatment of the syntactical sugar added is required, such as the new keywords (class, superclass, etc.).

Pseudo-variable super.

For example, the *reshape* in the class BorderedBox is a specialization of the *reshape* of the class Box, this requires binding the call 'reshape of super' to the right code in the super-class. Because only static binding is assumed, the treatment of the pseudo-variable super must be provided.

Pseudo-variable self.

In the example discussed, 'reshape of self' in operation *top* of class Box may deal with objects of class Box and BorderedBox - *top* is defined in Box and is inherited by class BorderedBox. This requires binding the call 'reshape of self' with the right code in the relevant object's class, i.e. the treatment of pseudo-variable self is needed.

In the following section two techniques will be presented to solve these issues.

3.4. Sub-classing implementation exploiting polymorphic procedures

The technique presented in this section exploits the polymorphic features of the procedure-based language considered. The technique is based upon the use of a new type called Dispatcher, which is a procedure type which takes as arguments a string of characters and a polymorphic type, and returns a polymorphic type as its result. In particular, a procedure of type Dispatcher takes the name denoting an operation as parameter, and returns either the results of the execution of that operation or a failure in the case the operation does not exist. In the implementation presented, every procedure which defines a class takes as its argument a procedure to access the new bindings defined by the calling environment, and returns an object of type Dispatcher. For example, object *bb1* of the class BorderedBox is created by the command:

Dispatcher *bb1* := *borderedBox*(*fail*);

where procedure *fail* specifies that no new bindings are defined, and is called when failures occur in binding an operation invoked upon *bb1*.

An example of inheritance implemented in procedure-based languages by means of the polymorphic type Dispatcher is illustrated by the programs of Figure 3.3 and Figure 3.4. The class BorderedBox is implemented by procedure *borderedBox*. Procedure *borderedBox* returns procedure *dispatch* which implements the dispatching strategy for BorderedBox, taking the name of the required operation as its argument. When executed, the body of procedure *dispatch* compares its argument with the names of the operations declared locally. When a match occurs the relevant procedure is called with the appropriate parameters; otherwise the *dispatch* procedure of the super-class is invoked with the same parameter. That is, at run time the search for an operation begins at the class of the invoked object, and proceeds to the top of the hierarchy - returning the first occurrence of the operation that is found. It is worthwhile to note that the code of *borderedBox* includes an object of class Box called *super* which allows the sharing of operations between a class and its super-class. This represents the implementation of the pseudo-variable super, and provides support for specialization within

```
% type of an operation %
type method = struct (  string  name,
                        poly    arguments );

% the type of class protocol %
type Dispatcher = proc (method)returns poly;

proc borderedBox = (Dispatcher bind) Dispatcher:
begin

    % superclass %
    Dispatcher super = box (newBinding);

    % instance variable %
    int borderSize = 2;

    % instance operations %
    proc reshape = (int newXOrigin, newYOrigin, newXLength, newYlength):
    begin
        eraseBorder;
        super ("reshape", (newXOrigin, newYOrigin, newXLength, newYlength));
        drawBorder

    end;
    proc setBorder = (int newBorderSize) ... ; % set a new border size %

    % private operations %
    proc eraseBorder = ... ;
    proc drawBorder = ... ;

    % new bindings required %
    proc newBinding = (method operation) poly:
    begin
        case bind(operation) in
        (fail):  if  nameofoperation = "reshape"  then reshape (argumentsofoperation)
                 else fail  % binding not here %
        out skip esac
    end;

    % declaration of the class protocol %
    proc dispatch = (method wanted) poly:
    begin
        if  nameofwanted = "reshape"      then reshape (argumentsofwanted)
        elif nameofwanted = "setBorder"   then setBorder (argumentsofwanted)
        else super (wanted)
        fi
    end;

    % return the class protocol %
    return dispatch

end.
```

Examples of creation and manipulation of instances of BorderedBox follow.

```
Dispatcher bb1 := borderedBox (fail);
Dispatcher bb2 := borderedBox (fail);
bb1 ("setBorder", 3);
bb1 ("move", (15,25));
bb2 ("reshape", (10,10,20,20));
bb2 ("top", (10,10,20,20));
```

Figure 3.3: Dispatching in class BorderedBox.

sub-classes, as can be appreciated by looking at the code of the procedure *reshape* in Figure 3.3.

For example, consider the command:

bb1 ("move", (15,25));

The *dispatch* procedure of *borderedBox* is invoked (via *bb1*) first with the string "move" and arguments (15,25) as its parameter. After having looked up the *move* operation among its local procedures and failed to find it, *bb1* calls the dispatcher of the super-class with the same parameter. This time the *move* operation is found (in class Box; see Figure 3.4), and applied to the parameters (15,25). (Note that in this code, no attempt has been made to optimize the run-time speed of the lookup in procedure *dispatch*, and a simple string search has been shown to emphasize the work of this procedure.)

The implementation for pseudo-variable self will be explained considering the execution of the command at the bottom of Figure 3.3

bb2 ("top", (10,10,20,20));

The search for operation *top* begins via procedure *dispatch* in class BorderedBox. The operation is not found in BorderedBox, so the search continues by looking in the super-class Box. An operation named top is found in Box (Figure 3.4); this operation contains in its body a case statement which causes the execution of procedure *newBinding* in BorderedBox with argument the operation name *reshape* - *newBinding* is the actual parameter of procedure *box* which is invoked in the code of BorderedBox to create object

```
% type of an operation %
type method = struct (   string  name,
                         poly    arguments );

% the type of class protocol %
type Dispatcher = proc (method)returns poly;

% box creates an instance of Box and returns the class protocol for the object %
proc box = (Dispatcher bind) Dispatcher:
begin
    % instance variable %
    int xLength  = 10;
    int yLength  = 30;
    int xOrigin  = 100;
    int yOrigin  = 200;
    shading defaultShade = white;

    % instance operations %
    proc move = (int newXOrigin, newYOrigin) ... ;
    proc reshape = (int newXOrigin, newYOrigin, newXLength, newYlength) ... ;
    proc shade = (shading newShade) ... ;
    proc draw = ... ;
    proc top = ... :
    begin
        - - -
        case bind ("reshape",(m,n,p,q)) in
            (fail): reshape (m,n,p,q)
        out skip esac;
        - - -

    end;

    % declaration of the class protocol %
    proc dispatch = (method wanted) poly;
    begin
        if  nameofwanted = "move"       then move (argumentsofwanted)
        elif nameofwanted = "reshape"   then reshape (argumentsofwanted)
        elif nameofwanted = "shade"     then shade (argumentsofwanted)
        elif nameofwanted = "draw"      then draw (argumentsofwanted)
        elif nameofwanted = "top"       then top (argumentsofwanted)
        else fail  % unknown operation %
        fi
    end;

 return dispatch

end.
```

Examples of creation and manipulation of instances of Box follow.

```
Dispatcher b1 := box (fail);
Dispatcher b2 := box (fail);
b1 ("move", (15,25));
b2 ("reshape", (10,10,20,20));
```

Figure 3.4: Dispatching in class Box.

super. The search for the *reshape* operation, therefore, begins in class BorderedBox. After that the call to procedure *bind* in *newBinding* fails (when object *bb2* was created, the actual parameter for *bind* was procedure fail), the relevant branch of the case statement (labelled by fail in Figure 3.3) is chosen, and an operation for reshaping is found and executed. It should be noted that this technique for implementing the pseudo-variable self adopts a depth-first search, that is first the class hierarchy is descended, and then the classes are visited proceeding to the top and searching for the first occurrence of the relevant operation.

In Figure 3.4, the procedure *box* for creating objects of the class Box is shown. Since the class Box has been declared without any super-class, procedure *newBinding* is missing, and a failure is caused by procedure *dispatch* if the operation being searched for is not present.

The technique discussed provides run-time support for sub-classing, and also solves the other issues discussed in the previous section.

3.5. Sub-classing implementation at objects creation-time

Unlike the technique in the previous section that binds operation names of an object at run-time, the technique presented in this section establishes all bindings at the outset during the object's creation; this is achieved by a careful use of procedure values and procedure variables. In the scheme presented below, every procedure which defines a class, say C, takes as its argument a parameter which holds the bindings redefined by the calling environment, and returns the set of operations to manipulate an object of class C. For example, object *bb1* of the class BorderedBox is created by the command:

BorderedBox *bb1* := *borderedBox* (*unbound*);

where *unbound* specifies that no new bindings are defined for object *bb1*, and is a variable of type BorderedBoxBind with the field bound set to false.

An example of inheritance implemented by exploiting this technique is illustrated by the programs of Figure 3.5 and Figure 3.6. The class BorderedBox is implemented by procedure *borderedBox* in Figure 3.5. The code of *borderedBox* includes an object of class Box called super which allows the inheritance of operations between a class and its super-class.

Procedure *borderedBox* returns together with the operations defined explicitly in class BorderedBox the object super. The invocation of an operation inherited by BorderedBox is achieved by prefixing the object name with the structure selector super. For example, the execution of the command

move **of** *super* **of** *bb1* (15, 25);

causes the procedure denoted by move in the field super of the structure *bb1* to be invoked and to be applied to the parameters (15,25). In other words, the operation *move* defined in class Box is executed on object *bb1* of class BorderedBox. The inclusion in the sub-classes of an object of the super-classes can be considered as the implementation of the pseudo-variable super, and provides support for specialization within sub-classes, as can be

```
type BorderedBox = struct (proc(int,int,int,int) void     reshape,
                           proc(int) void                 setBorder
                           Box                            super  );

% redefined bindings for type BorderedBox %
type BorderedBoxBind = struct (  bool                          bound,
                                 proc(int,int,int,int) void  reshape);

% borderedBox creates objects of class BorderedBox %
proc borderedBox ( BorderedBoxBind sub) BorderedBox:
begin

    % pass the redefined bindings to the superclass %
    BoxBind newBinding ;
    newBinding := (true, if boundofsub then reshapeofsub else reshape fi);

    % superclass %
    Box super := box (newBinding);

    % instance variable %
    int borderSize = 2;

    % instance operations %
    proc reshape (int newXOrigin, newYOrigin, newXLength, newYlength) :
    begin
        eraseBorder;
        reshape of super  (newXOrigin, newYOrigin, newXLength, newYlength);
        drawBorder

    end;
    proc setBorder (int newBorderSize) ... ;      % set a new border size %

    % private operations %
    proc eraseBorder ... ;
    proc drawBorder ... ;

    % return the class and the superclass protocol %
    return (reshape, setBorder, super)

end.


Examples of creation and manipulation of instances of BorderedBox follow.
BorderedBox bb1 := borderedBox (unbound);
BorderedBox bb2 := borderedBox (unbound);

setBorder of bb1 (3);
move of super of bb1 (15, 25);
reshape of bb2 (10, 10, 20, 20);
top of super of bb2 (10, 10, 20, 20);
```
Figure 3.5: Implementation of class BorderedBox.

appreciated by looking at the code of the operation *reshape* in Figure 3.5.

The implementation for pseudo-variable self is based upon the use of a structure type having name of the form <class name>Bind, e.g. BorderedBoxBind. The first field of this type is a boolean and indicates whether the remaining fields have been bound to values. The remaining fields specify the operations which in the declaration of a class or of its super-classes appear to be invoked via the pseudo-variable self. For example the type BorderedBoxBind is defined as:

type BorderedBoxBind = **struct**(**bool** *bound,*
proc(int,int,int,int)void *reshape*);

Since the pseudo-variable self has not been used in the code of *borderedBox*, BorderedBoxBind contains the same operation of BoxBind.

The operations invoked via pseudo-variable self are implemented by employing procedure variables. This approach allows the binding of the operations invoked via self with the relevant procedure bodies at the time objects are created. For example, the call of the *reshape* operation by the *top* operation declared in Box can be bound to the code which is able to manipulate objects of class BorderedBox at the time such objects are created. Let us explain how this is achieved by considering the execution of the command:

BorderedBox *bb2* := *borderedBox* (unbound);

To set up all the bindings for object *bb2*, the execution of procedure *borderedBox* and of procedure *box* is required. During the execution of procedure *borderedBox*, which is shown in Figure 3.5, first the tuple (true, *reshape*), where the name reshape denotes the *reshape* procedure declared in *borderedBox*, is assigned to the variable newBinding, and second the result of the execution of the procedure *box* with argument newBinding is assigned to variable super. This second assignment also causes the execution of procedure *box* which, as can be appreciate by looking at the code in Figure 3.6, binds the operation to reshape objects of class BorderedBox to the procedure variable denoted by *reshapeOFself*. This binding will be used at run-time to call the operation relevant to manipulate the object *bb2*.

In order to show that all the bindings established for object *bb2* are sufficient to support sub-classing, consider the execution of the command at the bottom of Figure 3.5

top **of** *super* **of** *bb2* (10, 10, 20, 20);

The operation *top* in class Box (Figure 3.6) is executed with argument (10,10,20,20). This operation contains in its body a procedure variable, denoted by *reshapeOFself*, which causes the execution of operation *reshape* in BorderedBox - the operation to reshape BorderedBox has been passed as actual parameter to procedure *box* and has been assigned to procedure variable *reshapeOFself* when the object *bb2* was created. It is worthwhile to note that this implementation for the pseudo-variable self does not rely upon any run-time lookup - the bindings for operation names of an object can be established at the outset during object's creation and do not need to be modified during the object's lifetime.

```
type Box = struct(  proc(int,int) void           move,
                    proc(int,int,int,int) void   reshape,
                    proc(shading) void           shade,
                    proc void                    draw
                    proc(int,int,int,int) void   top );

% redefined bindings for type Box %
type BoxBind = struct( bool                        bound,
                       proc(int,int,int,int) void  reshape);

% box creates an instance of Box and returns the operations to manipulate it %
proc box (BoxBind sub) Box:
begin
    % instance variable %
    int xLength  = 10;
    int yLength  = 30;
    int xOrigin  = 100;
    int yOrigin  = 200;
    shading defaultShade = white;

    % instance operations %
    proc move (int newXOrigin, newYOrigin) ... ;

    % procedure to reshape boxes %
    proc reshape (int newXOrigin, newYOrigin, newXLength, newYlength) ... ;

    proc shade (shading newShade) ... ;
    proc draw ... ;

    % variable declaration to hold the procedure reshape redefined by the sub-class %
    proc(int,int,int,int) void reshapeOFself;
    reshapeOFself := if boundofsub then reshapeofsub else reshape fi;

    proc top ... :
    begin
        . . . reshapeOFself(m,n,p,q); . . .
    end;

 return (move, reshape, shade, draw, top)

end.
```

Examples of creation and manipulation of objects of Box follow.

```
Box b1 := box(unbound);
Box b2 := box(unbound);

move of b1 (15, 25);
reshape of b2 (10, 10, 20, 20);
top of b2 (10, 10, 20, 20);
```

Figure 3.6: Implementation of class Box.

In Figure 3.6, the procedure *box* for creating objects of the class Box is shown. Some examples of box manipulations are also shown. This time the invocation of the *top* operation, that is

top **of** *b2* (10, 10, 20, 20);

causes the execution of the code to reshape objects of class Box. When *b2* is created, the argument of procedure *box* is unbound (a variable of type Box-Bind with the field bound set to false), and the procedure *reshape* declared in box is assigned to the procedure variable *reshapeOFself* by the command

reshapeOFself := **if** *bound* **of** *sub* **then** *reshape* **of** *sub* **else** *reshape* **fi**;

This simple mechanism allows the binding of operation *reshape* differentiating between the code defined in class Box and that defined in class BorderedBox. It must be emphasized that because the class Box has been declared without any super-class, procedure *box* does not include an object denoted by super among the values returned as results.

3.6. Preprocessor specification

The techniques discussed above for implementing sub-classing in a procedure-based language permit the writing of object-based programs. However, although they allow class hierarchy and inheritance to be implemented, most object-based constructs are hidden behind procedure manipulations.

A further step towards providing a more convenient programming context would be to implement a preprocessor which deals with a program containing explicit class and sub-class declarations and produces the code for the procedure-based compiler. Note that such a preprocessor is not required for the enforcement of any specific discipline, but only to make the syntax more convenient. This represents a major difference with respect to other approaches which implement a class hierarchy and inheritance as a set of extensions to a procedural language. For example, the C++ preprocessor implements object encapsulation, while the approach in this chapter provides object encapsulation by exploiting first-class higher-order procedures, which is a feature of the base language.

In the case of the technique presented in Section 3.4 such a preprocessor could take a program in the notation of Figure 3.2 and transform it to that of Figure 3.3 and Figure 3.4. As can be appreciated by comparing these figures, the preprocessor should perform the following steps:

1. Fix the syntactical discrepancies, namely substituting the keywords proc for class, and the occurrences of the type Dispatcher for any class types.

2. Modify the program to deal with pseudo-variables super and self. The actions to be carried out in this step include appending to each class description the code for procedure *dispatch* and for procedure *newBinding*, and for returning *dispatch* as a value. The code for *dispatch* and *newBinding* can be generated automatically by parsing the declaration of the structure of the operations the classes should return.

3. Identify the type of the formal polymorphic parameter of the procedure *dispatch* for each class C. This type can be modelled as the union of the types of the formal parameters of all the operations declared in C and in its super-classes. Besides, *dispatch* may also return values of a type which is the union of the types of the results of the operations declared in C and in its super-classes. These unions can be computed during the parsing of the operation declarations in phase 2.

4. Change all the procedure calls of the kind 'id1 of id2(parameter-list)' where id2 is a class identifier in the command 'id2(id1, (parameter-list))'.

In the implementation for the Flex system, the identification of the unions in phase 3 was not needed. The Flex architecture provides specific machine instructions which have been exploited for the treatment of the polymorphic parameter of the *dispatch* procedure. Note also that the preprocessor discussed can convert operation names into some representation that can be compared efficiently to avoid the string comparison overhead in the body of the *dispatch* and *newBinding* procedures.

In the case of the technique presented in Section 3.5, as can be appreciated by comparing Figure 3.2 with Figure 3.5 and Figure 3.6, the steps the preprocessor should perform include:

1. Modify the program to deal with pseudo-variables super and self. The actions to be carried out in this step include declaring in each class

with a super-class definition the object super, returning it as a value, and declaring a procedure variable for each operation invoked via self.

2. Identify for each class C the type of the formal parameter, called <C>Bind, and the type of the value returned (the class operations exported) by the procedure defining C. The code for the declarations of these types can be generated automatically for each class by parsing the declaration of the structure of the operations following the keyword instance operations.

3. Insert in every operation call the right sequence of super. In order to minimize the overhead needed to compile the operation calls, it is convenient for the preprocessor to maintain a table recording for each operation the distance between the class inheriting it and that defining it. This data structure can be initialized during the parsing of the operation declarations in phase 2.

If the definition of compilation units, or packages [DoD80], is supported by the compiler, the data structure at step 3 can also be exploited to minimize the number of packages to be recompiled after an update of the class hierarchy. When a class, say C, is updated, the preprocessor can trace the packages which contain invocations to C's operations and need to be recompiled. Keeping the code of each operation in separated packages is a good strategy to minimize the recompilation overhead. Incidentally, it should be noted that any type errors in the operation calls will be detected at compile-time.

3.7. Performance considerations

Measurements have been made of the overhead caused by these schemes. The measurements were made on an ICL Perq workstation running the Flex system, and have been carried out for a null body operation call. The average time has been measured for such a call to be completed.

With respect to the technique of Section 3.4, it was found that the results of the measurements have a very close approximation to the following expression:

*n * (time for dispatch call) + (time for operation call)*

where *n* is the number of the classes visited to find the relevant operation. The invocation of an operation not inherited is composed of two procedure calls: a call to the *dispatch* procedure, and a local call (internal to the class and invoked by the *dispatch* procedure). In general, to call an inherited operation, the time needed to execute the *dispatch* procedure has to be multiply by the number of classes visited.

The performance degradation due to the scheme of Section 3.4 for *n* equal to one (operation not inherited) is of the order of 35%. This performance degradation is by comparison with an operation call involving a reference to a class object. The call of any operation of class Box, as defined in Figure 3.1, involves a reference to a variable (e.g. *b1*); such an indirect call of an operation (via a variable holding the class object) is needed for data abstraction regardless of the inheritance scheme proposed. Although the degradation increases linearly with the distance of the inherited operations (as shown by the above expression), given that an object-based program can be expected to spend only a modest fraction of its time actually invoking operations, the overall performance degradation should be at the very most of the order mentioned above. The search down the hierarchy for supporting the implementation of the pseudo-variable self has the same cost: a linear number of calls to procedure *bind* plus a local call.

The performance degradation due to the scheme of Section 3.5 for calling an inherited operation from a direct super-class is of the order of 1%. This performance degradation is by comparison with a non-inherited operation, and is caused by the indirection in calling an inherited operation, namely via the variable super holding the super-class object. The implementation of the pseudo-variable self has a smaller cost - regardless of the distance of the operation definition, its call is performed via a procedure variable involving one level of indirection only.

3.8. Concluding remarks

Two general techniques have been presented for implementing the subclassing form of inheritance as a set of extensions to a high-level language. The first technique exploits the polymorphic features of the language to implement the operations inheritance of a class hierarchy, while the second technique exploits first-class procedures.

To a certain extent, the choice between these techniques depends on what object-based programming is being used for. If interest is essentially in quick prototyping and experimentation, the first approach may be the best to adapt. Keeping the dispatching scheme during execution makes it easier to change the class hierarchy in order to correct errors, improve the system, or experiment with new facilities - there is no need on each occasion to go through a complete compilation of other classes in the hierarchy. On the other hand, if the concern is for correctness, robustness and efficiency, then the second approach, which constructs most of the bindings at compile-time, is obviously required. For example, correctness requires that one knows what the system is before one executes it; if the system can be changed during its execution, there is little hope that one can guarantee any of its properties.

The merits of these two techniques include: (1) they are useful as a discipline for achieving object-based programming in a non object-based programming language, (2) they suffice for providing run-time support for sub-classing in procedure-based languages, and (3) they preserve the static binding and the strong type-checking features of the languages, hence obtaining - with high degrees of correctness, testability, and maintainability - the benefits of inheritance discussed in Chapter 2. Moreover, the techniques proposed in this chapter show that there is no need to always adopt run-time lookup to provide sub-classing, although such strategy is employed by many object-based languages, such as Smalltalk-80. Instead high level languages with static binding and strong type-checking can be exploited.

Other approaches to adding sub-classing to a procedural base language include Objective C [Cox86], and Extended Pascal [Jacky87]. Both of these approaches are based upon a message-passing mechanism. In Objective C, the message mechanism is centralized in a single function called the messager. Objects contribute a dispatch table that the messaging routine searches to determine how this kind of object implements its operations. This table is built when the object is defined, and is looked up dynamically. It identifies every operation that this object knows how to perform, and a pointer to a procedure whereby this object implements that operation. This procedure is a compiled C function body.

In Jacky's approach [Jacky87], operations are invoked through messages passed by placing them on a last-in-first-out message stack. The dispatcher mechanism in this case examines the top message on the stack, determines which class of object it is being sent to, and calls an invoke procedure that includes the operations for that class. This methodology provides inheritance in a very limited sense only.

An interesting conclusion to be drawn from this study is that the family of languages considered allows high degrees of freedom in programming and does not restrain the programmer within the procedure-based approach. An object-based style of programming is also possible, and although procedure-based languages do not constrain one to this programming discipline, thanks to the proposed technique, there appears to be no inherent reasons for the programmer of such languages to prefer one approach over the other.

The rest of this thesis will only be concerned with object-based programming, whether this is achieved with an explicit language or by means of the techniques for using procedure-based languages described in this chapter. In particular, in the following chapter, the applications of the object-based methodology to the design and implementation of distributed architectures will be reviewed.

Chapter 4

Distributed Object-Based Architectures

In recent years a great deal of interest has developed in distributed object-based systems and architectures. Distribution gives rise to some issues that do not exist in a centralized design, or that exist in a less complex form. For example, distribution forces high costs on the sharing of data and code, and fault tolerance techniques for a centralized system (which is either running or crashed) are simple by comparison to those of a distributed system (which may be partly running and partly crashed).

Many of the issues concerning the design and implementation of distributed architectures, whether object-based or not, are issues to do with reliability. This chapter, tries and summarizes what these issues are, in particular and whenever possible describing them in terms of object-based programming ideas. Problems that cause distributed architectures to be unreliable are pointed out, and different approaches that have been proposed for attaining reliable distributed processing are reviewed.

4.1. Distribution issues

Object-based architectures can be described as a set of classes, each of which can be thought of as representing a kind of resource. Some resources may have a direct physical realization, such as I/O device. Others may have a logical realization, such as processes, mailboxes (for communication between processes), and files. With object-based architectures, each individual resource is an object; thus an object encapsulates the resource implementation and provides a set of operations, these operations being the only means by which the object can be manipulated. In most *distributed object-based architecture* (see [Liskov82, Svobodova84, Almes85, Birman85, Tanenbaum86, Shrivastava88a] for a representative sample), an operation is performed by invoking an operation of the object with a remote procedure call (RPC), which passes value parameters to the object and returns the results of the operation to the caller.

Before analyzing the features of distributed object-based architectures, let us focus on the issues caused by distribution. The key to understanding distribution issues is an appreciation of the *logical* and *physical partitioning* of components within a distributed architecture. The very essence of a distributed architecture consists of physical partitioning at some level, possibly of several components partitioned around a computer room and connected by a local area network, or across a continent and connected by a wide area network. All logical partitionings will be motivated by or will have to take into account this physical partitioning. In other words, the concept of partitioning forces the system designer to answer two questions. The first is how should the effects of partitioning be reflected in the applications. Indeed, partitioning requires: (1) explicit communications between interacting components, since no storage is shared, and (2) global management strategies and policies. The second question is how should the partitioning be exploited to achieve desirable quality attributes. Partition may be exploited to provide: (1) isolation as a method of enforcing security and safety policies; (2) tolerance of independent component faults and recovery from such faults without disruption of the whole system; (3) truly parallel execution; and (4) incremental growth or contraction of a system, through the addition or subtraction of discrete components.

Most designers believe that the effects of partitioning should be transparent to applications. Tanenbaum suggests that a distributed architecture should appear to be *a virtual uniprocessor* rather than a collection of individual computers [Tanenbaum85]. If this ideal is achieved in practice, an architecture may be described as having *distribution transparency.* Such a distributed architecture conceals the consequences of distribution from applications and users; that is, the architecture is perceived as a whole rather than a collection of independent resources - users need not be aware of which component executes their programs or stores their files. Amoeba [Tanenbaum81] is an example of such distributed architectures.

Distribution transparency groups together various strategies, abstractions and mechanisms, which can be better understood, if the specific aspects of transparency are discriminated. Below, some basic forms of transparency are briefly presented; for a more thorough treatment, see [ANSA87].

Access transparency

Access transparency conceals the communications services from users, such that invocations on objects are semantically and syntactically identical whether the objects are local or remote.

Failure transparency

Failure transparency is the property of an architecture that enables the full concealment of faults despite the failure of components. In most distributed object-based architectures, this feature is obtained by executing programs which operate on objects as *atomic actions* with respect to failures [Gray78]. The *failure atomicity* property of the atomic actions ensures that a computation can either be terminated normally, producing the intended results, or be aborted producing no results. This property may be obtained by appropriate use of *backward error recovery,* which is invoked whenever a failure that cannot be masked occurs. It is also convenient that once an atomic action terminates normally, the results produced are not destroyed by subsequent node crashes. This property, called *permanence of effect,* ensures that state changes produced are recorded on stable storage which can survive node crashes with a high probability of success. A two-phase commit protocol is required during the termination of an action to ensure that either all the objects updated within the action have their new states recorded on stable storage (normal termination), or no updates get recorded (abnormal termination).

Concurrency transparency

Concurrency transparency allows parallel use of an object without the concomitant emergence of inconsistent views of that object. Concurrent executions of object operations are free from interference if the *serializability property* is ensured - a concurrent execution can be shown to be equivalent to some serial order of execution [Eswaren76, Best81]. A variety of concurrency control techniques to enforce the serializability property have been reported in the literature [Bernstein87]. A very simple and widely used approach is to regard all operations on objects to be of type read or write, which must follow the well-known locking rule permitting concurrent reads but only exclusive writes.

Migration transparency

Migration transparency allows the movement of objects without making such transfers apparent to other objects. This functionality leads towards strategies for global optimization. For example, it can be exploited to reconfigure the distributed architecture to optimize performance, or to restart after failures.

In the following sections, the basic issues related to the provision of the four forms of distributed transparency presented above will be discussed, thus identifying the software mechanisms for controlling and exploiting distribution.

4.2. Access issues

One widely discussed framework for communication is the ISO OSI reference model, which has seven protocol layers [Zimmerman80]. By using this model it is possible to connect heterogeneous networks composed of computers running widely different operating systems. Unfortunately, the overhead created by all these layers is substantial. In a distributed architecture consisting primarily of computers connected by slow leased lines, the overhead might be tolerable. Plenty of computing capacity would be available for running complex protocols, and the narrow bandwidth available implies that close coupling between the nodes would be impossible anyway. On the other hand, in a distributed architecture consisting of identical microcomputers connected by a 10-megabyte per second or faster local network, the price of the ISO model is generally too high without special hardware support.

The model favoured by most researchers for distributed architectures is the *client-server model* in which a client process requiring some service sends a request message to the server and then waits for a reply message. The basic primitives in the simplest form of client-server model are *send* and *receive.* The send primitive specifies the destination and provides a message to be sent; the receive primitive specifies the source of the message and provides a buffer where the incoming message is to be stored.

A more structured form of communication is achieved by distinguishing requests from replies. As will be discussed in the next subsection, with this approach, communication in message-passing systems appears very similar to a traditional procedure call from the client to the server.

4.2.1. Remote procedure calls

The remote procedure call (RPC) model [Nelson81, Birrell84] has become an accepted method, mainly because of its distribution transparency and straightforward translation into client-server interactions. The idea is to make the syntax and semantics of internode communication as similar as possible to local procedure calls within the application program's high-level language, because such procedure calls are familiar and well understood, and have proved their worth over the years as abstraction tools.

To a first approximation, an RPC scheme works in the following way: the client (caller) and the server (callee) modules are programmed as if they were intended to be linked together. A description of the server interface, that is, the names of the procedures and the types of arguments the server implements, is processed yielding two *stubs,* where the stubs deal with translating procedure calls into appropriate message interchanges. The client stub is linked with the client; to the client this stub looks like the server. The server stub is linked with the server; to the server this stub looks like the client. The stubs shield the client and server from the details of communication. In an RPC execution: the client issues a normal procedure call on its node with the intention of invoking a procedure of the server. It actually issues a call to the client stub running on its own node, as shown in Figure 4.1. The client stub collects the parameters and packs them into a message which is then sent to the server stub at the remote node. Afterwards, the client stub blocks, waiting for a reply. The server stub, on receiving a request message, unpacks the parameters and invokes a local procedure call on the server. The results of the local procedure call follow an analogous path in the reverse direction.

This approach is attractive in many ways. For example, it achieves access

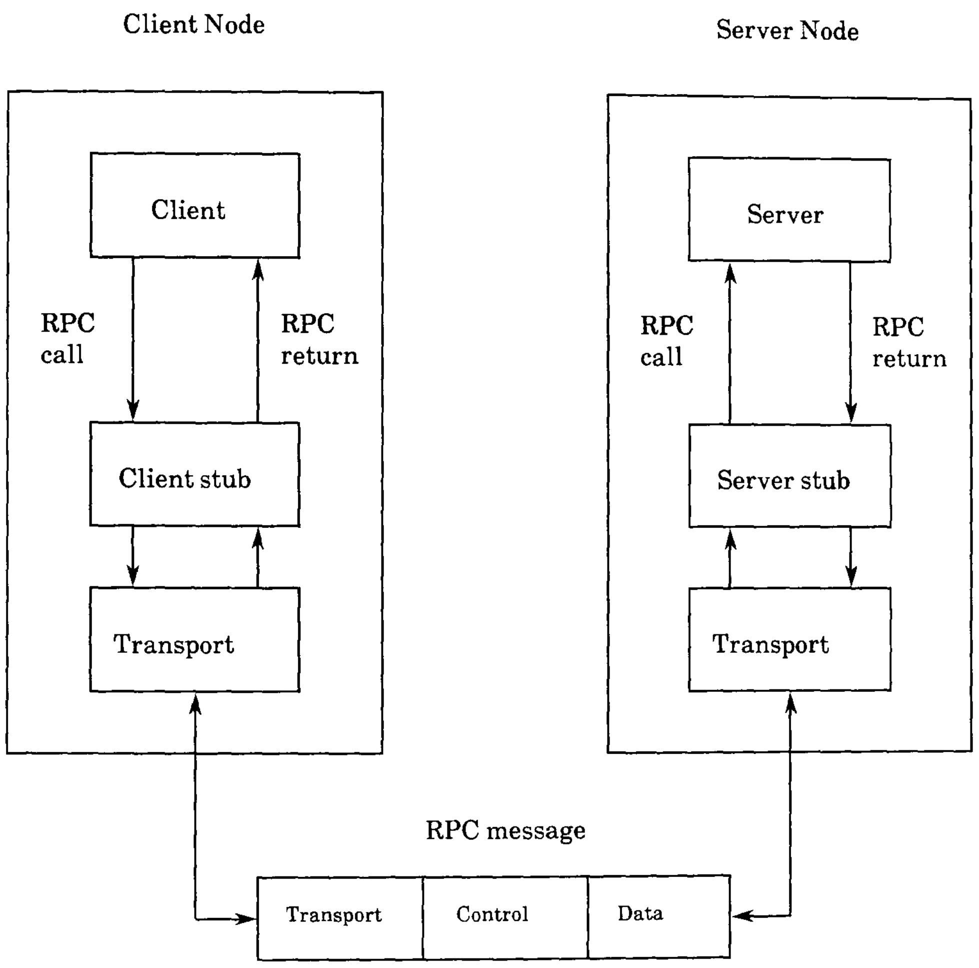

Figure 4.1: Remote procedure call.

transparency. The client need not know anything about the fact that the server is remote - it just issues an ordinary procedure call with the parameters passed in the usual way, such as on the stack. Similarly, the server is called by a local procedure according to local calling and parameter passing conventions. However, a number of reliability-related problems hide under the surface. Many of these have important implications for the system's overall reliability.

In general, the ideal of having the semantics of an RPC identical to that of a local procedure call is hard to achieve, if communication and node crashes occur. In the interest of reliability, a client process may retransmit its request message whenever the loss of that message is suspected. As a result, it is possible for a server occasionally to receive multiple request messages for a single invocation by the client. Therefore, unless preventive measures are employed, the server could carry out the same request several times. These superfluous and undesirable executions are referred to as *orphan* executions [Nelson81]. Orphan executions can sometimes cause problems of reliability and consistency. For example, if a bank server has to transfer a large amount of money to a Swiss bank account, someone would prefer that operation not be executed by accident a second time. Operations that can be carried out multiple times without harm, such as reading a block of some file, are said to be *idempotent*. Unfortunately, most operations that cause actions to occur in the outside world, and involve communication or I/O operations, are not idempotent. Various mechanisms to detect and prevent orphan executions will be discussed in the next section.

4.3. Failure issues

A major concern in the design of distributed architectures is to provide continuous service of the system as a whole in spite of node crashes and communication failures. This section considers these sources of unreliable behaviour in distributed architectures, focusing in particular on node crashes, due either to hardware or software.

In the previous section, it has been shown that unreliable communications can cause orphan executions. Another possibility of orphan executions can arise with node crashes. To appreciate this, consider the following situation: a computation running at node B issues an RPC to some object at node A and then node B crashes, leaving an orphan computation running at node A. If the client process at B resumes that call, after recovery, by reissuing it, the concurrency that might arise between the post-crash call of B and the orphan computation at A should be regarded as undesirable, since it is expected that the execution of a sequential program should give rise to a sequential computation characterized by a single flow of control. Therefore, in the presence of client failures, an RPC mechanism ought to guarantee also that the executions of all post-crash calls of the client follow all pre-crash ones. What is required, at least, is that node A should detect and abort the orphan before executing the post-crash call from node B. Nested RPCs make the scenario more complex - see [Panzieri85] for a detailed discussion of the issue. It should be noted that the orphan execution as in the discussed example might arise even if messages are never lost.

Various methods have been proposed for dealing with orphans [Nelson81, McKendry85, Panzieri88]. One obvious and unacceptable method is to kill off all processes in the entire distributed system when any node crashes. This is, of course, unacceptable, but it is just what many conventional implementations do when a vital process fails.

Several methods of killing orphans have been proposed in [Nelson81]. One of them exploits clock synchronization. If the nodes have synchronized clocks, then orphans can be killed simply by establishing a time limit on each RPC. More precisely, an *expiration time* is associated with each process. Servers inherit their expiration times from their clients. In this way, the caller sets the time limit for all its servers. Whenever a process reaches its expiration time and is still executing, it is declared an orphan and promptly aborted.

The method presented in [McKendry85] also uses an expiration-based mechanism to eliminate orphans created by crashes and aborts. The method is based on clocks local to each node, but it performs best when clocks are synchronized.

The orphan treatment method proposed in [Panzieri88] employs local, crash-proof, clocks. In addition to a *deadline* mechanism that resembles the Nelson's expiration time, every node maintains a variable *crashcount* which is the local clock value at the time the node was rebooted after a crash. A node also maintains crashcount values of clients which have made calls to it. A newly created server checks the client-supplied crashcount value against the corresponding value maintained at the node; if the former is the greater, then this indicates that the caller has experienced a crash, in which case there could be orphans on the node. The server then aborts all other servers created by that client. To cope with a failed client that does not recover, every node runs a *terminator* process. Such a process regularly constructs a list of potential orphans (that is, servers that do not receive requests for a few minutes) on its node and calls relevant clients to see if they are still running. If a client is not running its correspondent server processes are aborted.

So far, the reliability problems caused mainly by clients and communication failures have been considered. In general, servers also can fail, hence producing an interruption of the service provided. The provision of continuous service is achievable by the use of redundant components. In distributed architectures there is an opportunity to keep redundant copies and to provide surplus processing resources.

One common way of classifying the redundancy employed in distributed architectures is to differentiate *active redundancy* from *passive redundancy*. Redundancy may be active in the sense that all the redundant components are operating simultaneously. Alternatively redundancy may be passive in the sense that only one component is in service and the others are in stand-by mode.

A well-known passive redundancy technique is backward error recovery. The objective of backward error recovery is to restore a computation to a state prior to the manifestation of a fault [Anderson81a]. As backward error recovery restores a presumably valid prior state of a program, it is an attractive technique that can be used to provide recovery after all types of faults, even unanticipated faults in the software design. Thus, backward error recovery is often used to meet the failure atomicity requirements of the operations in object-based systems.

There are several situations in which the passive redundancy approach clearly does not suffice, and active redundancy is required. These include situations where the frequency and duration of recovery time are unacceptable, or where the continuity of correct I/O behaviour must be ensured, such as in flight control systems. Active redundancy techniques include the canonical N-Modular Redundancy [Wensley78, Mancini86a], where the client always sends call requests to all replicated servers, and performs majority voting on the results. So long as the majority of the servers is non-faulty the correct result will be chosen. In such a scheme all the non-faulty servers must be kept in the same state. This requirement is particularly hard to meet in distributed architectures, and its violation can lead to the so-called *sequencing failures,* as discussed in [Mancini86b]. Active redundancy can also be employed to tolerate unanticipated faults in the software, as shown by the N-version programming technique [Avizienis84]. With this technique, N different versions of a program are executed at the same time. Their results are compared and the result which is submitted by the majority of the versions is chosen.

The topic of fault tolerance in distributed architectures will be expanded in

Chapter 5, where it will be shown that fault-tolerant schemes which happen to have been developed within the domain of object-based architectures can be mapped and applied with profit to more conventional architectures.

4.4. Concurrency issues

One of the reason for choosing a distributed architecture is to take advantage of the potential concurrency in an application, thereby increasing efficiency and decreasing response time.

The concurrent use of objects requires control mechanisms in order to prevent the emergence of non-serializable executions. Despite the fact that many different techniques which regulate the use of shared objects have been proposed, most new distributed systems, such as Argus [Liskov87], use locking as concurrency control mechanism. A *lock* is a serialization mechanism which ensures that only one operation accesses the object at a time. It has the effect of notifying others that the object is busy, and of protecting the lock holder from modifications of others. A simple lock protocol associates a lock with each object. Whenever an object is used, the client follows a two-phase locking policy to overcome the problem of non-serializable executions [Eswaren76]. The idea is to divide the acquisition and release of locks into two distinct phases. During the first phase, termed the growing phase, locks can only be acquired and not released. In the second phase, the shrinking phase, locks may only be released and new ones may not be acquired. In addition, in order to avoid the problem of *cascade* aborts, it is necessary to make the shrinking phase instantaneous. That is, suppose that an action in its shrinking phase is to be aborted, and that some updated objects have been released. If some of these objects have been locked by other actions, the abortion of the action will require these actions to be aborted as well.

However, the two-phase locking concurrency control can restrict the degree of permissible concurrency in an application. It is typical for an application to require nested invocation of operations. In such a case, locks on objects acquired by inner operation are retained by the outermost operation until the outermost operation is itself committed. This means that objects can remain unavailable to other clients for a long time, thus introducing a potential performance problem. What is required is a controlled means of introducing *internal concurrency,* i.e. concurrency inside an object.

Several research projects have been studying the design and implementation of distributed object-based architectures from the point of view of introducing internal concurrency. The most relevant among them include: Argus [Liskov87], Actors [Agha86], and ABCL/1 [Yonezawa86].

Although different terminology is used, these three architectures share several properties. All of them are based on objects, though these are called *guardians* in Argus and *actors* in Actors, which provide several approaches to concurrency. Concurrent processing among objects is supported via non-blocking invocation of objects. Moreover, there are opportunities for concurrency internal to an object. When an operation of an object is invoked, it causes an independent thread of control (e.g. a lightweight process) to be activated within the invoked object. The details of process activation, and the degree of and control over concurrency within an object differs for each of the three architectures mentioned above. The differences in internal concurrency can be described in terms of a model, where each object employs one or more servers to process a queue of invocation requests.

In Argus, the abstraction of an unbounded number of servers for each object is provided. Requests are serviced (i.e. servers are created) immediately and there actually is no queue. Synchronization and control (e.g. server blocking and serialization) are accomplished via locks which are shared by all of an object's servers.

An Actors object can also have an unbounded number of servers, but its servers are controlled quite differently. Initially, an actor has a single server, which processes a single message and dies. Sometimes during its lifetime, it creates a successor to process the next request on the queue. Early creation of successors is permitted and this allows the use of multiple servers. There are no locks, there is no data sharing between servers, and each server can have different semantics.

An ABCL/1 object employs a priority interrupt single server. The processing of a request can be interrupted to handle a high priority request, or the next message on the queue. Interrupt processing and persistent variables allow multiple requests to share the environment of a single server.

Argus, Actors, and ABCL/1 represent different approaches to the use of objects to facilitate the design and implementation of concurrent systems.

Some of the differences are dramatic - compare a action-based, heavyweight, multi-threaded object such as a guardian in Argus, to a short-lived, lightweight object running on one processor in a massively parallel computer. Each approach has engineered the object-based methodology to solve a rather different problem, each with its own performance, resource sharing, and expressibility trade-offs. Indeed, it appears that good engineering is the dominant theme in the design of concurrent and distributed architectures, and that objects are a useful tool for this engineering process.

4.5. Migration and reconfiguration

An important reason for wanting a distributed implementation is its potentiality for adding and reconfiguring hardware resources to increase processing power, decrease response time, or increase availability of data. It is also important to move software components in order to provide recovery from failure, to balance the load across the nodes, and/or to improve the efficiency of a particular application. For example, if a client sends several requests to the same remote object, the overhead of transmitting messages may become high. It might be better to move the object from the remote node to the local node so that future operation invocations would occur locally.

It is then fundamental to implement distributed architectures that support software migration. Migration allows systems to be reconfigured dynamically, by adding and removing components, or by moving components from one node to another while the system continue to operate. To minimize the impact of moving objects, the method used to invoke operations must be location independent. Mechanisms are required to determine if an object exists on a node, and if so to provide its address.

Several research have focused particularly on the problem of *migration* of objects within a network of computer nodes, for example the Emerald system [Black86]. The location protocol employed in the Emerald system for translating addresses when an object moves is discussed in [Fowler85]. A survey of the most innovative approaches for migration transparency is presented in [Smith88].

4.6. Concluding remarks

Distributed architectures are characterized by the physical partitioning of their components. This partitioning, which requires explicit communications between different physical components, introduces a number of fundamental issues concerning the visibility of distribution.

Various forms of distribution transparency have been identified in [ANSA87]. They can be regarded both as problems to be solved in order to conceal the partitioning of architectural components, and as features to be exploited to take advantage of the partitioning in order to achieve particular levels of security, reliability, and performance. This chapter has carried on from the analysis in [ANSA87] to discuss various strategies, abstractions, and mechanisms required for controlling and exploiting distribution in object-based architectures.

Just as Chapter 2 on object-based programming was followed with a chapter illustrating a means of achieving the same effects without using an explicit object-based programming language, so the next chapter will go on to discuss work which illustrates the relationship between distributed object-based architectures and an apparently different form of distributed architectures, based on processes.

Chapter 5

Object-Based versus Process-Based Distributed Architectures

The many approaches to fault-tolerant systems incorporating error recovery reveal various similarities and differencies. In this chapter, two canonical architectures for distributed fault-tolerant computing are constructed and shown to be duals of each other. One architecture incorporates objects and actions as the entities for program construction while the second architecture employs communicating processes with checkpoints. As a consequence of the duality, techniques which have been developed within the domain of just one of the architectures can be mapped and applied to the other.

In the following sections, the essential aspects of object-based and process-based architectures are first described. Next, the arguments intended to establish the duality are pointed out. Finally, the usefulness of the fault-tolerance duality concept are illustrated, by mapping some well-known object replication techniques developed within the context of the objects and actions architecture to the communicating process architecture thereby revealing some interesting process replication techniques.

5.1. The canonical architectures

An investigation of fault tolerance techniques employed in a variety of systems reveals a partitioning into two broad classes. Two canonical architectures are proposed, each embodying the major characteristics of the corresponding class of systems. The first architecture incorporates objects as the entities for program construction while the second architecture employs communicating processes. One widely used technique of introducing fault tolerance - particularly in distributed systems - is based on the use of *atomic actions* (atomic transactions) for structuring programs [Gray78]. An atomic action possesses the properties of serializability, failure atomicity and permanence of effect. Atomic actions operate on objects. The class of applications where such an *Object Model* (OM) has found usage include banking, office information, and database systems. A number of other applications - typically concerned with real time control - are structured as concurrent processes communicating via messages. Some examples are process control, avionics and telephone switching systems. Fault tolerance in such systems is introduced through a controlled use of *checkpoints* by processes. This way of constructing an application will be referred to as employing the *Process Model* (PM).

This chapter, which is a revised and extended version of work reported earlier in [Shrivastava88b, Mancini89], claims that the OM and PM approaches to the provision of fault tolerance are duals of each other and presents arguments and examples to substantiate the claim. As a result of this observation, it can be stated that there is no inherent reason for favouring one approach over the other; rather the choice is largely dictated by the architectural features of the underlying layer. Indeed, one would now claim that the differences between the two approaches are basically a matter of viewpoint and terminology. The investigations presented have been influenced by the well-known duality paper of Lauer and Needham [Lauer78] which puts forward the notion that within the context of operating systems, procedure-based systems and message-based systems are duals of each other. Lauer and Needham observed that (1) a program or subsystem constructed strictly according to the primitives defined by one architecture can be mapped directly into a dual program or subsystem which fits the other architecture; (2) the dual programs or subsystems are logically identical to each other, and they can also be made textually very similar; and (3) the performance of a program or subsystem from one architecture will be identical to its counterpart. The present work may be considered as an extension of the ideas put forward in that paper with regard to fault tolerance.

5.1.1. The object-based architecture

One of the most important aspects of the OM architecture is that objects and actions are the two primary entities from which an application program is constructed. Any atomic action can be viewed at a lower level as constructed out of more primitive atomic actions - this is illustrated in Figure 5.1 which also introduces the action diagram which will be used in this chapter, this notation is based on that used by Davies [Davies73]. According to Figure 5.1, action B's constituents are actions B_1, B_2, B_3 and B_4. A directed arc from an action (e.g. A) to some other action (e.g. B) indicates that B uses objects released by A.

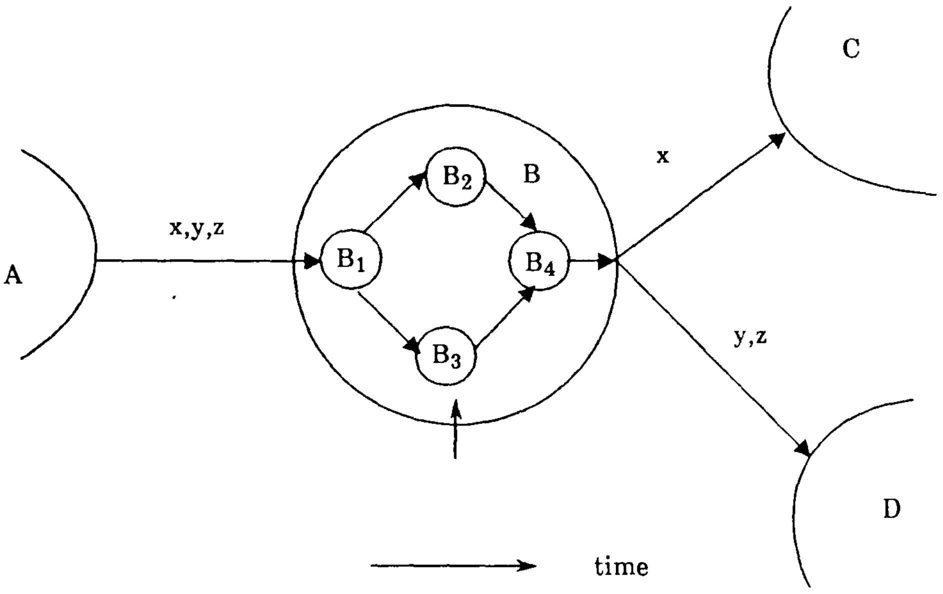

Figure 5.1: Action diagram.

Optionally, an arc can be labelled, naming the objects used by the action. In Figure 5.1, B uses objects x,y and z, and C uses object x which has been released by B. Actions such as B_2 and B_3 are executed concurrently. Nested actions give rise to nested recovery. Suppose time has advanced up to the point shown by the vertical arrow, and an error is detected in B_3 causing it to be aborted. What happens after B_3's recovery? The question must be resolved within the scope of B - the enclosing action. B can provide a specific exception handler to deal with this particular eventuality, such exception handling techniques have been discussed by Taylor [Taylor86]. If no handler is available, then a failure of B_3 causes B to be aborted.

Any implementation of actions and objects will require processes (clients and servers) for carrying out the required functions. However, the role played by processes is hidden at the application level. Similarly, there is no explicit use of message passing between entities, since RPCs hide the details of message interactions between clients and servers. For example, in the Argus programming system [Liskov82], the implementation of guardians (objects) requires a number of processes for receiving and executing calls from clients - but processes are not visible entities to be used explicitly by an application program. Taylor [Taylor86] describes a number of ways of implementing atomic actions using different process structures. In the OM architecture, objects are long lived entities and are the main repositories for holding system states, while actions are short lived entities.

5.1.2. The process-based architecture

In contrast to the OM architecture, where processes and messages play a secondary role, the PM architecture uses them as the primary entities for structuring programs. An application is structured out of a number of concurrent and interacting processes.

The PM architecture will be assumed to have the following characteristics: (1) processes do not share memory, at least explicitly, and communicate via messages sent over the underlying communication medium; (2) appropriate communication protocols ensure that processes can send messages reliably such that they reach their intended destinations uncorrupted and in the sent order; (3) a process can take a checkpoint to save its current state on some reliable storage medium (stable storage). If a process fails, it is recovered back to its latest checkpoint.

In a system of interacting processes, the recovery of one process to its checkpoint can create an inconsistent global state, unless some other relevant processes are recovered as well. This leads to the notion of a consistent set of checkpoints or a recovery line [Randell78]: a set of checkpoints, one from each process, is consistent if the saved states form a consistent global state. Figure 5.2 illustrates the notions of consistent and inconsistent sets of checkpoints where opening square brackets on process axes indicate checkpoints and sloping arrows represent messages. Suppose process p fails at the point indicated by the vertical arrow and is recovered back to its latest checkpoint. The global state of the system as represented by the set of checkpoints on the cut C_2 is inconsistent since the checkpoint of r has recorded a message which has not yet been sent by p; the set of checkpoints on recovery line C_1 is however consistent. Thus a failure of p can cause a cascade recovery of all the four processes - this is the domino effect mentioned in [Randell75]. The dynamic determination of a recovery line is a surprisingly hard task; the reader should consult [Wood81, Koo87] for a clear exposition.

The domino effect can be avoided if processes coordinate the checkpointing of their states. A well-known scheme of coordinated checkpoints is the conversation scheme [Randell75, Banatre78, Wood81, Koo87]. The set of processes which participate in a conversation may communicate freely between each other but with no other processes. Processes may enter the

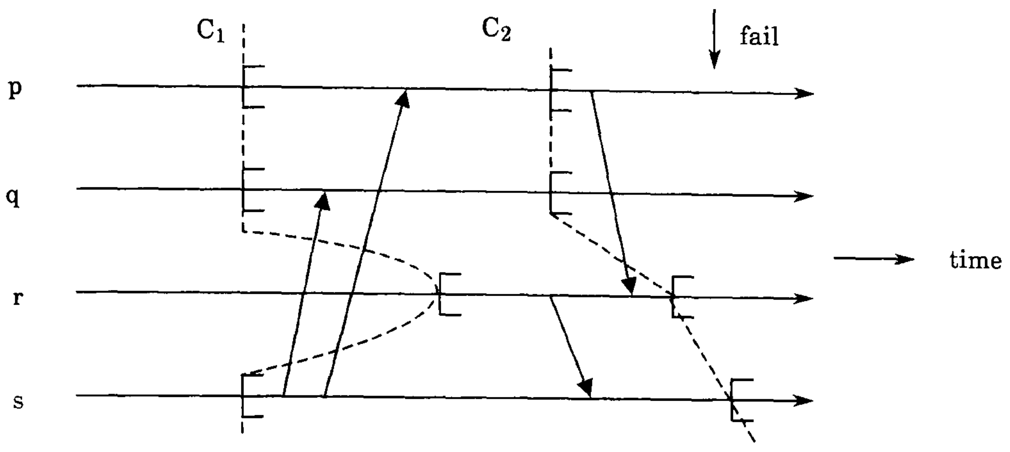

Figure 5.2: Consistent and inconsistent sets of checkpoints.

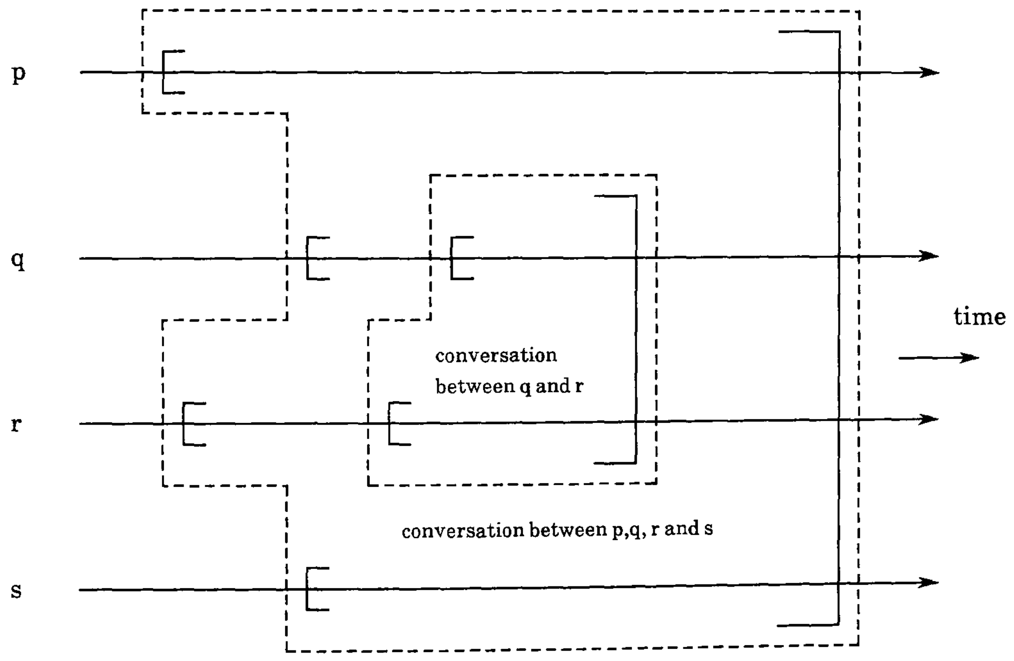
p
q
r
s
time
conversation between q and r
conversation between p,q, r and s

Figure 5.3: Conversations.

conversation at different times but, on entry, each must establish a checkpoint (see Figure 5.3). In Figure 5.3, a closing bracket indicates that all participating processes must exit at the same time after taking fresh checkpoints (brackets will not be explicitly drawn in the subsequent diagrams). If a process within a conversation fails then all the participating processes are recovered back to the respective checkpoints established at the start of the conversation. Conversations can be nested, as indicated in the figure.

Conversations provide a convenient structuring concept for introducing fault tolerance in a large class of real time systems [Anderson81b]. The need to respond promptly to changes in the external environment dictates that most real time systems have an iterative nature. The PM architecture provides a natural way of expressing such systems in the form of interacting cyclic processes with synchronization points usually associated with timing constraints. A study of real time system structure for avionic systems by Anderson and Knight [Anderson81b] indicated that synchronization of processes in such a system stems from the need to synchronize with the events in the external environment, rather than from any inherent needs of processes themselves.

The most important aspects of the PM architecture relevant to the duality mapping are summarized below. An application is programmed in terms of a number of processes interacting via message passing. If processes establish checkpoints in an arbitrary manner then there can be a danger of cascade recovery, which is usually undesirable. Conversations provide a coordinated means of managing checkpoints to avoid the danger of such a cascade recovery. However, a conversation requires the participating processes to synchronize such that they exit from the conversation simultaneously. A large class of applications, typically concerned with process control or real time control, traditionally employs the PM architecture for structuring applications. Conversations can be imposed on such applications by exploiting naturally occurring synchronization points among interacting processes. In the PM architecture, processes are long lived entities and main repositories for holding system states, while conversations are short lived entities.

5.2. The duality argument

The canonical architectures discussed in the previous sections are representative of the corresponding class of fault-tolerant systems. Given a description of any fault-tolerant system, it is usually straightforward to work out its representative architecture, despite the fact that the terminology used for the description may even differ some what from that used here. The duality between the OM and PM architectures can be established by considering objects and actions to be the duals of processes and conversations respectively. Further, object invocations can be considered duals of message interactions [Lauer78]. A given conversation diagram (e.g. Figure 5.4a), can be translated into an action diagram quite simply (e.g. Figure 5.4b) by replacing each conversation C_i with a corresponding action A_i, and adding an arrow from A_i to A_j if C_i and C_j have at least one process in common and that process enters C_j after exiting from C_i. An arc from one action to the other is labelled with the objects representing the processes common to the

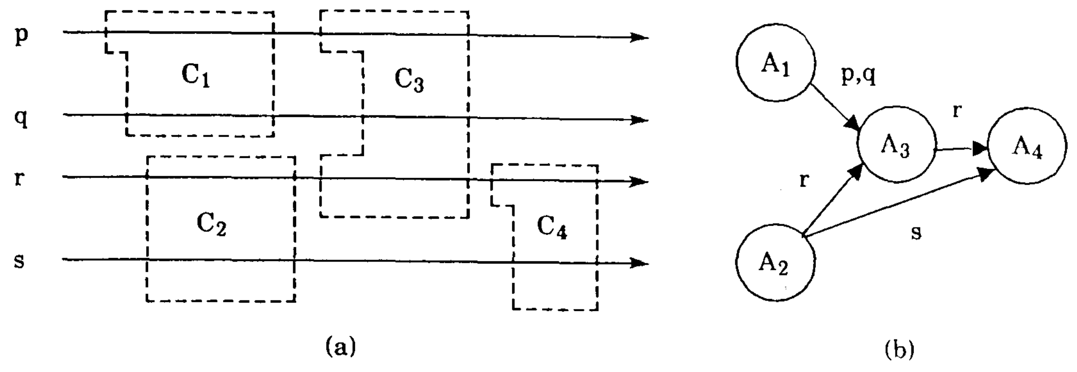

Figure 5.4: Conversations and actions.

corresponding conversations. A reverse mapping is possible by replacing distinct objects named in the action diagram by processes. An action is replaced by the corresponding conversation determined by the set of objects named in all the incoming and outgoing arcs of the action.

In order to support the hypothesis, it will be discussed the way in which three major properties of a fault-tolerant computation, namely, (1) freedom from interference, (2) backward recovery capability, and (3) crash resistance, are embodied in the OM and PM architectures.

1. Freedom from interference. In the OM architecture, this requirement is ensured by the serializability property of actions and enforced by some concurrency control technique, such as two-phase locking. In the PM architecture, freedom from interference between multiprocess computations structured as conversations is ensured by the two conversation rules, (i) a process can only communicate with those processes that are

in the same conversation; and (ii) a process can only be inside a single conversation at a time (this rule can be relaxed under certain conditions, see later). The two-phase locking discipline for actions corresponds to entering a conversation (growing phase) and leaving a conversation (shrinking phase).

2. Backward recovery capability. An action in progress can be aborted (recovered) without affecting any other ongoing actions. This recovery property of an action is enforced in conjunction with the concurrency control technique in use. In the case of two-phase locking, this means that all the held locks are released simultaneously. This corresponds to the synchronized (simultaneous) exit from a conversation which is required from all the participating processes. The act of taking checkpoints at the start of a conversation has its dual in the OM architecture, and consists of the requirement of maintaining recovery data for objects used within an action. It was indicated earlier that the serializability property of actions can be maintained even if - for two-phase locking - locks are released gradually (rather than simultaneously) during the shrinking phase of locking; however this has the danger of cascade aborts (recovery of an action can cause some other actions to be aborted as well). A similar observation can be made for conversations: the synchronized exit requirement is necessary to prevent cascade aborts. Figure 5.5 illustrates that if 'conversations' C_1 and C_2 do not observe the rule of synchronized exit, and if time has advanced up to the point shown by the vertical arrow, and C_1 is to be aborted, then C_2 will have

to be aborted as well.

3. Crash resistance. A two-phase commit protocol is employed in OM to ensure that despite the presence of failures such as node crashes, an action terminates either normally, with all the updated objects made stable to their new states, or abnormally with no state changes. A similar protocol will be required to ensure that the states of all the processes participating in a conversation are made stable.

A striking benefit of establishing the duality is that the body of knowledge and techniques developed for one architecture can be mapped and applied to the other architecture. This is illustrated with the help of the following two examples.

Read only requests

A number of optimizations are possible if an action uses some or all of its objects in read only mode. Read locks can be released during the shrinking phase and need not be held till the end of the action, without the danger of cascade aborts. Further, no recovery data need be maintained for read only objects and they need not be involved in the two-phase commit protocol since they do not change state. Such optimization strategies have been studied extensively within the context of database systems, see for example [Mohan83]. However, no such strategies have been studied for conversations, although they can be developed quite easily. Essentially, processes inside a conversation that do not update their states need not synchronize their exit from the conversation, nor do they need to take checkpoints at the

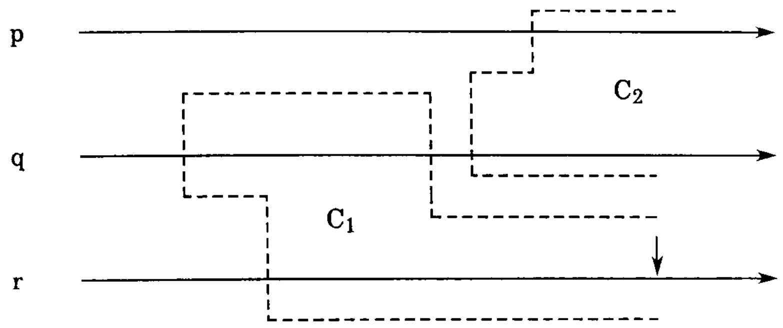

Figure 5.5: Cascade aborts.

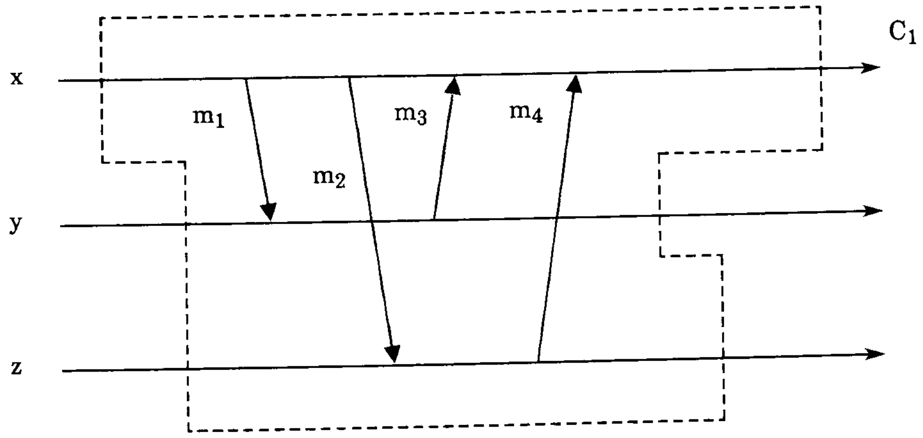

Figure 5.6: Read only requests.

start of the conversation. Consider a simple example. An action performs the following computation: x:=y+z. Here y and z will be read locked; the commit protocol will involve only making object x stable to its new state and the action need generate no recovery data for y and z. Figure 5.6 shows a possible conversation to perform the same computation. In this particular case it is only necessary for process x to establish a checkpoint. Message m1 (m2) is a request to y (z) for some value, and message m3 (m4) contains the value sent by y (z).

Note that even though there is a two way exchange of messages between x and y (z), x can recover without affecting y (z), since message m1 (m2) is a read request. Indeed, y and z can take part in other conversations, while still in C_1, provided those conversations also involve only read requests directed to y and z. This is obviously the dual of the shared read lock mode rule applicable in the OM architecture. It is worthwhile to note that, just as locking can cause deadlocks among actions, similar problems can occur in conversations.

Programmed exception handling

So far the duality has been examined from the point of view of backward error recovery, which involves abandoning the current state for a prior state. In contrast, forward error recovery involves selective corrections to the current state to obtain an acceptable state [Randell78]. Programmed exception handling is a means of incorporating this form of forward recovery. A widely accepted exception handling strategy is as follows: if during the execution of a computation an error is detected (an exception is detected) for which a specifically programmed handler is available, then that handler is invoked; if there is no programmer-provided handler available then a default handler is invoked whose function is to invoke backward recovery. Thus, exception handling can provide a uniform means of incorporating both forward and backward error recovery strategies [Anderson81a, Cristian82]. A recent paper [Campbell86] proposes an exception handling strategy for concurrent processes with conversations and describes how processes can resolve concurrent exceptions through the use of exception trees. It is worthwhile to note here that these exception handling ideas, although developed using the PM architecture, have since been applied by Taylor [Taylor86] to the OM architecture.

A summary of the various characteristics of the two architectures for which duality has been established is presented in Figure 5.7.

5.3. Some examples

This section contains two further examples, one taken from the database area and normally programmed using objects and actions and the other taken from the process control area and normally programmed using processes and conversations. It will be shown that programs written using the primitives of one architecture have duals in the other. Simple and self-explanatory notation will be used for program description.

Process Model	*Object Model*
Processes	Objects
Conversations	Actions
Message interactions	Object invocations
Conversation rules preventing no outside communication	Concurrency control for serializability
Stable processes	Stable objects
Processes entering a conversation	Growing phase (two-phase locking)
Processes leaving a conversation	Shrinking phase (two-phase locking)
Read only request messages	Read locks

Figure 5.7: Duality mapping.

Banking application

An example often used to illustrate the properties of an action concerns transferring a sum of money from one bank account to another. The failure atomicity property will ensure that either the sum of money is debited from one account and credited to the other, or no state changes are produced. For the sake of illustration, the application has been structured to invoke nested actions, even though simpler, non-nested solutions are clearly possible.

Two classes of objects will be assumed: Standing-order, and Credit-debit. Their definitions are given by the program of Figure 5.8, which also shows the creation of objects *order, acc1* and *acc2*.

An invocation of 'transfer of order(...)' will give rise to a nested computation as shown in Figure 5.9. Any exceptions during the execution of *transfer* will cause that action to be aborted.

The same program can be recoded quite easily in terms of communicating processes, as shown in Figure 5.10. A transfer conversation can be initiated by sending a message to the *order* process. The transfer conversation is shown in Figure 5.11.

Process control application

The second example is taken from a process control application in the coal mining industry [Sloman87]. A pump installation is used to pump mine-water collected in the sump at the shaft bottom to the surface. The pump is enabled by a command from the control room. Once enabled, it works

```
class Standing-order;
                  - - object variables - -

   action transfer (credit-debit to, from; dollars amount)
                  cobegin
                  authority (to, from);
                  credit of to (amount);
                  debit of from (amount)
                  coend
   end action
                  - - other actions, e.g. authority - -
end Standing-order;


class Credit-debit;
                  - - current account variables - -

     action credit (dollars amount)
                  - - add amount - -
     end action

     action debit (dollars amount)
                  - - subtract amount - -
     end action
                  - - other actions - -
end Credit-debit;


Standing-order order;
Credit-debit acc1, acc2;
```

Figure 5.8: Example of banking program in OM.

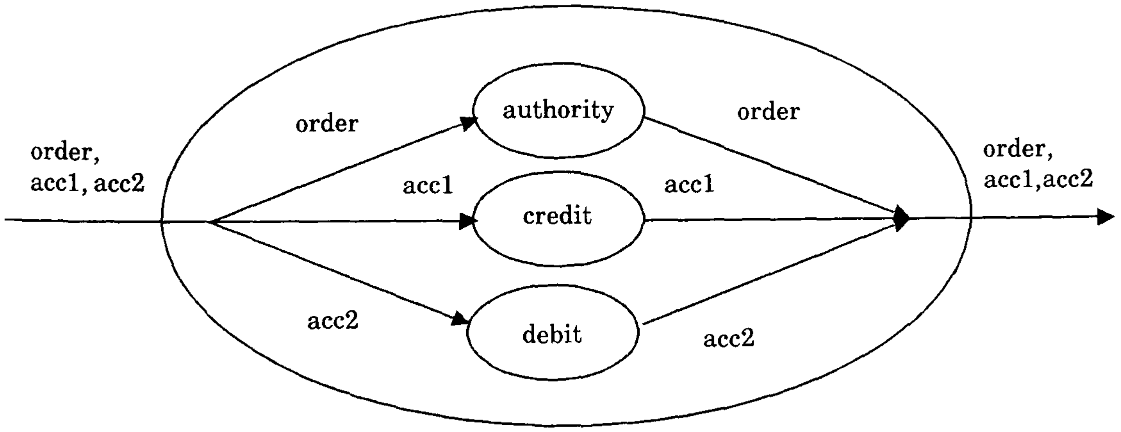

Figure 5.9: A bank action.

```
task type Standing-order;
                    - - process variables - -
   select
     conversation transfer (credit-debit to, from; dollars amount)
                    cobegin
                    send (self, authority, to, from);
                    send (to, credit, amount);
                    send (from, debit, amount)
                    coend
     end conversation
   - - other selections, e.g. authority - -
   end select
   end Standing-order;

   task type Credit-debit;
                    - - current account variables - -
   select
     conversation credit  (dollars amount)
              - - add amount - -
     end conversation
   - - other  selections, e.g. debit - -
   end select
   end Credit-debit;

   Standing-order order;
   Credit-debit acc1, acc2;
```

Figure 5.10: Example of banking program in PM.

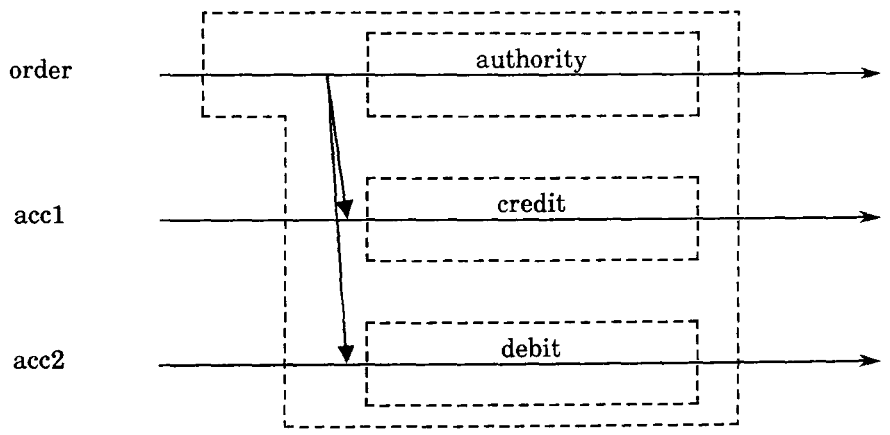

Figure 5.11: A bank conversation.

automatically, controlled by water level sensors; detection of a high level causes the pump to run until a low level is indicated. For safety reasons, the pump must not run if the percentage of methane exceeds a certain safety limit. Some other parameters of the environment are also monitored by the monitoring station.

The control software can be structured as five communicating processes, namely: Pump-controller, Surface, Level, Pump and Monitor. Some sketchy details are given here for the Pump-controller.

The functions of the Pump-controller process are to receive start/stop command from the Surface process (representing the control room), receive water level reports from the Level process and to receive an alarm signal from the Monitor process. The Pump-controller process can send start/stop commands to the Pump process which controls the pump.

A study of process structure discussed in [Sloman87] reveals that the overall behaviour of the other processes have a similar structure to the Pump-controller, either receiving requests to carry out certain functions and/or sending messages to other processes to request certain functions to be performed. These interactions can be organized as conversations. A simplified program fragment for the Pump-controller is given in Figure 5.12.

A command to enable or disable the pump from the Surface process starts a conversation containing the Pump-controller and the Pump process: if the conversation terminates normally, the pump will have changed state accordingly. It is fairly easy to reprogram this example in terms of objects and

Process Model	*Object Model*
task type Pump-controller;	**class Pump-controller;**
- - *process variables* - -	- - *object variables* - -
select	
conversation *on/off*(...) *send start/stop command* *to the pump process* **end conversation**	**action** *on/off*(...) *call start/stop command* *of the pump object* **end action**
- - - *other selections* - - -	- - - *other actions* - - -
end select	
end Pump-controller;	**end Pump-controller;**

Figure 5.12: Pump-controller example.

actions, with the five processes replaced by the corresponding objects. For the sake of illustration, the program for the Pump-controller class is also shown in Figure 5.12.

These examples provide further empirical support to the duality claim by illustrating that close similarity exists between the two classes of programs. Given a program constructed from the primitives defined by one architecture, it can be mapped directly into a dual program of the second architecture.

5.4. An application of the duality mapping

By realizing the duality between the OM and PM approaches to fault tolerance, it is possible to map techniques developed within the context of one architecture to the other architecture. Section 5.2 showed how optimization techniques for read only actions can be applied to conversations. In the rest of this chapter, replicated process management techniques will be developed from replicated object management techniques. More precisely, the following problem is considered. Given is a set of concurrent processes interacting via message passing. It is assumed that a subset of these processes (server processes) provide various services to the remaining processes (client processes) and these services have to be made available despite a bounded number of server node crashes. Thus it is necessary to replicate servers on different nodes. Algorithms for implementing such a system where client processes interact with replicated servers within the framework of conversations have not been studied before. Here it will be shown how such algorithms can be developed easily by applying the duality mapping to object replication techniques which have been studied extensively.

5.5. A review of object replication techniques

In a system where nodes never fail, replicated objects can easily be managed. It is sufficient to perform any operation of an object x on all copies of x. Unfortunately, this approach is impractical in systems where nodes can fail and recover. For example, this approach requires that each operation be performed on all copies of x, even if some have failed. Since there will be times when some copies of x are down, the system will not always be able to perform the required operation on all copies of x at the time it receives the request. If the system were to adhere to this approach, it would have to delay processing the operation until it could access all copies of x.

Such a delay is obviously unsatisfactory. If any copy of x fails, then no action that invokes x can execute to completion. The more the copies of x, the higher the probability that one of them is down. In this case, replication actually makes the system less fault-tolerant.

Several techniques have been proposed to manage replicated objects. To be specific two well-known techniques will be considered: the available copies [Bernstein84] and the primary copy schemes [Alsberg76, Stonebraker79]. In the following, these techniques will be briefly described. It is assumed that nodes fail in a fail-silent manner (that is, a node is either operational or down, it does not suffer byzantine failures), and that all operational nodes can communicate with each other. Therefore, each operational node can independently determine which nodes are down, simply by attempting to communicate with them. If a node does not respond to a message within some timeout period, then it is assumed to be down.

5.5.1. The available copies scheme

As stated earlier, an object provides a set of operations, some of which can modify the state of the object (e.g. push and pop operations of a stack object). Initially, it will be assumed that a node does not recover after a failure.

The available copies scheme does not require an action to update all copies of each object. An action should send every operation request to all of the copies that it can, but it may ignore any copies that are down. After sending an operation request to all copies of object x, an action may receive rejections from some nodes (if the operation is conflicting with some other action), positive response from others (meaning the operation has been accepted and performed), and no response from others (those that have failed). Operation requests for which no responses are received are called missing. If any rejection is received or if all operation requests to x's copies are missing, then the whole operation is rejected and the action must abort. Otherwise, the whole operation is successful. Since a fail-silent behaviour of the nodes is assumed, anyone of the positive responses can be taken as the result of the operation invocation.

5.5.2. Recovery

So far it has been assumed that a failed node does not recover; it is also possible to provide the system with some reconfiguration mechanism, in order to support the recovery of a copy from a failure, and in general the creation and removal of copies of an object. To achieve reconfiguration after a node crash, the set of nodes holding the available copies of an object must be dynamically established.

A solution is to employ directories to record for each object x the set of x's copies that are available. Like any other object, a directory may be replicated, that is, it may be implemented as a set of directory copies at different nodes. In the following discussion, it is assumed that there is a fixed set of copies for each directory, known to every node. That is, new directory copies are never created - the method for creating new object copies can be easily extended to create new directory copies.

Directories recording available object copies are manipulated by two special actions, Join for creating new object copies, and Disjoin for deleting unavailable object copies. When a node N containing a copy of x, say x[N], recovers from a failure, the system runs an action Join(x[N]). Join(x[N]) brings the state of x[N] up-to-date by: (1) finding a directory copy d listing the set of copies of x; (2) reading d to find an available copy of x, say x[M]; (3) copying x[M]'s state into x[N]; (4) declaring x[N] to be available by making an entry for x[N] in each available copy of the directory d.

When a node fails, some client that tries to invoke an object operation at that node observes the failure. The system, then, runs a Disjoin action for each copy stored at the failed node. Disjoin declares the relevant object copy to be unavailable by removing the entry for this copy from every available copy of the directory.

To process an operation with this recovery scheme, the system reads a copy of a directory and issues the operation for every copy of the object x that the directory says is available. If the system discovers that any copy that the directory says is available is actually unavailable, the system runs a Disjoin(x) action. Because a recovering node uses a Join action for bringing the states of its replicated objects up-to-date, there is no need for such objects to use stable storage.

5.5.3. Read optimization

Optimizations are possible with the available copies scheme if the semantics of the operations is taken into account. For example, the operations exported by each object may be partitioned into two classes: *Write,* which comprises the operations that modify the state of the object, and *Read,* which comprises the operations that do not alter the state of the object.

The operations of *Read* do not need to be invoked on all available copies of an object but just to one, while the operation requests of *Write* need to be sent to all available copies of an object. For example, in the case of the stack object, the operation top, which returns the value at the top of the stack without modifying the stack, can be invoked just on any available copy of the stack. The operations push and pop are of class Write, and must be sent to all available copies of the stack.

A distributed two-phase locking scheme can be employed for concurrency control. The following rule is required: whenever an operation of *Read* is invoked on an object, the action must first acquire a read lock (if not already acquired) on any available copy of the object; for a write operation, the action must first acquire write locks (if not already acquired) on all the available copies of the object.

With such an optimization, the available copies scheme may lead to problems of correctness. There will be times when some copies of an object x do not reflect the most up-to-date state of x. An action that uses an out-of-date copy of x can create an incorrect, i.e. non-serializable, execution, even if only failures, but not recoveries occur. To avoid this well-known problem (see [Bernstein87] for details), a validation protocol is required. An action's validation protocol starts after its operations on copies have been acknowledged or timed out. At that time the action knows all the copies it has actually accessed. The validation protocol makes sure that all copies that were unavailable (available) are still unavailable (available).

The read optimization with validation protocol scheme suffers from the limitation that in certain situations an action has to be aborted if a failure occurs. As has been discussed previously, without read optimization the completion of an action can be guaranteed in the presence of a specified number of node failures, by distributing operation requests to all available copies. Distributing information about read requests, and in particular about read locks, may seem unreasonably expensive. However, in [Birman85], a scheme for lazy propagation of read locks is mentioned which guarantees that read lock information is delivered to a node before any action that requires this information is executed.

The Join and Disjoin actions discussed previously are still required to support recovery with read optimization. In the subsequent discussions, the term pure available copy scheme will be used to refer to the particular scheme without read optimization.

5.5.4. The primary copy scheme

With the primary copy scheme, executing actions use a non-replicated view of the system. That is, for each object that the actions access, the operations are carried out on the same copy of the object, called the primary copy. The distribution of the operations to other backup copies is delayed until the action has terminated and is ready to commit. It is necessary therefore to maintain an intentions list of deferred operations. During the termination of an action, the appropriate portion of the intentions list has to be sent to each node that contains backup copies of the relevant objects. Alternatively, the primary copy of an object can send its new state in place of the intentions list. If the primary copy fails then the executing action is aborted and can be resubmitted to use a different copy that will take over as primary.

In order to support recovery after a failure of a primary copy, it is necessary to elect a backup copy as the new primary. A simple scheme, that does not involve additional communication, is to determine *a priori* the order of selecting the copy to use next. An alternative is to run a consensus protocol among the backup copies - the election of the new primary copy can be based on the current load on the system. The level of availability can be maintained after a failure by running an action to create a new backup copy.

With the primary copy scheme, it is possible to put all deferred operation requests destined for the same node in a single message. This tends to minimize the number of messages required to execute an action. By contrast, with the available copies scheme, the action sends operation requests to replicated copies while it executes. Thus the available copies scheme tends to use more messages than the primary copy scheme. Another advantages of the primary copy scheme is that aborts often cost less compared with the available copies scheme. In the available copies scheme, when an action aborts, it is likely that many of the action's operations have already been distributed to replicated copies. Not only are these operations wasted, but they must also be undone. With primary copy, the distribution of those operations are delayed until termination time making abortion cheaper. Fast aborts in the primary copy scheme are at the expense of commits which can be more time consuming than in the available copies scheme. This is because during the first phase of the commit protocol a node may be asked to process a potentially large number of deferred operations on backups.

With the primary copy scheme, read optimization is possible - the intentions lists of only write operations need be distributed to backup copies when the executing action is ready to commit.

The most important aspects of the object replication techniques relevant to this discussion are summarized below. The pure available copies scheme provides k-object-resiliency, meaning that out of k copies of an object, all the k copies have to become unavailable before the action using it is forced to abort. With read optimization, k-object-resiliency is not always reachable; this is the price paid for obtaining higher efficiency. The primary copy scheme does not provide k-object-resiliency in the sense mentioned above; the executing action has to be aborted if the primary fails. The action can be resubmitted once a secondary is elected to be the primary.

5.6. Process replication techniques

Process replication techniques take advantage of the existence of multiple processors by replicating critical processes on two or more nodes.

A terminology commonly used for classifying the redundancy employed in PM is to differentiate active redundancy from passive redundancy. With an active redundancy scheme, a given computation is executed simultaneously on a number of processes, while with a passive redundancy scheme, if the process running the computation fails then a designated backup process takes over. Not surprisingly, active redundancy techniques correspond to the available copy schemes and passive redundancy techniques to the primary copy schemes.

The duality mapping between object and process replication schemes is shown in Figure 5.13.

5.6.1. The available processes scheme

The dual of the available copy scheme results in an approach where replicated processes behave like a single process. Interactions with a replicated process implies interactions with all of its replicas. A copy of a request to a replicated process is sent to all replicas, and all replicas execute each request. In case a reply is required, all replicas generate replies; only the first reply received is considered, and the others are discarded (since the replies from all working replicas should be identical under the fail-silent assumption on processor behaviour). A reconfiguration strategy for the

	Active replication			*Passive replication*		
	pure form	recovery	read optimization	pure form	recovery	read optimization
Object replication	available copy	available copy+join	available copy+ validation	primary copy	primary copy+ election	only writes in intentions list
Process replication	available process	available process+ join	available process+ validation	primary process	primary process+ election	only state changes in intentions list

Figure 5.13: Object and process replication.

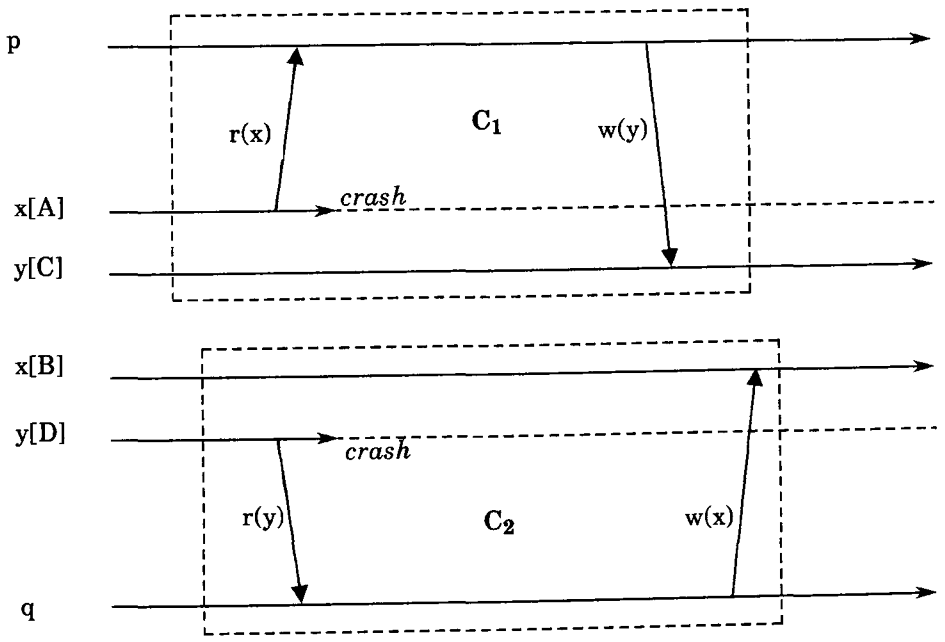

Figure 5.14: Incorrect execution.

available processes scheme can be designed by adopting the directory based scheme. Note that replicated processes need not record their checkpoints on stable storage (see Section 5.5.2).

5.6.2. Read optimization

Read optimization can be achieved in PM using the approach employed in OM. Assume that processes receive message via message ports which are data structures capable of holding messages of a certain class. Message ports can be of class *Read,* capable of receiving messages whose processing does not alter the state of the receiver process, and *Write,* capable of receiving messages whose processing can alter the state of the process. A message intended for a read port of a process need not be sent to all available copies of that process.

Read optimization in PM results in weakening of the conversation rule. Instead of having all the copies of a replicated process participating in only one conversation at a time, a replicated process can take part in more then one conversation, if only requests of *Read* are being served.

This optimization may lead to problems of consistency similar to those encountered in OM, as can be appreciated by considering the process diagram of Figure 5.14. Here a system with non-replicated processes p and q and replicated processes x (copies x[A], x[B]) and y (copies y[C], y[D]) is considered. Suppose that within the conversation C_1, p reads the state of x and updates the state of y, denoted by r(x) and w(y) respectively, and that, within the conversation C_2, q reads the state of y and updates the state of x, denoted by r(y) and w(x).

Conversation C_1 begins by reading from x[A], and conversation C_2 begins by reading from y[D]. After p and q complete their reads, processes x[A] and y[D] fail. Then p and q perform their writes. Since y[D] is down, y[C] is the only available copy of y. So C_1's w(y) is invoked only on y[C]. Similarly, since x[A] is down, C_2's w(x) is invoked only on x[B].

The execution above does not violate the conversation rule for read optimization (just as in OM, the two-phase locking rule will not be violated). However, this execution is not equivalent to any serial execution. A serial execution of C_1 and C_2 on a non-replicated system would have either C_1 reading the value of x written by C_2, or C_2 reading the value of y written by C_1 - in the example neither conversation reads the data written by the other.

The dual of the validation protocol is required at the end of conversations to ensure correctness. The aim of the validation protocol in the PM is to make sure that all processes found unavailable (available) during the execution of a conversation are still unavailable (available).

5.6.3. The primary process scheme

With the primary process scheme, the same copy of a replicated process, called primary process, takes part in conversations. A replicated process is provided with backup processes on different nodes (in particular, just one backup process might be employed). Request messages are sent to the primary process, which handles the requests. The distribution of requests to other backup processes is delayed until the end of the conversations. At that time, the primary copy sends the list of requests served during the conversation to the other backup processes. Alternatively, the primary process can send a checkpoint of its new state to the backup processes. In the event of a primary process failure, the executing conversation is recovered to the beginning and restarted with a backup process which takes over and become primary. The dual of the election scheme mentioned earlier will be required to select the new primary process.

5.7. Some existing process replication schemes

Schemes following the available processes approach have been proposed in the literature independent of the data replication techniques [Schneider87, Cmelik88]. In [Schneider87], a general approach is proposed for coordinating copies of replicated processes so that each copy executes the same sequence of process interactions. This is achieved by the implementation of the abstractions of agreement and order. A similar approach is adopted in [Cmelik88], where client processes send request messages to all the copies of a replicated server, and a distributed consensus protocol for every request is employed to enable each copy of the server to process requests from different clients in the same order. However, these papers do not describe how aborts are performed and domino effect avoided if a client recovers back. Furthermore, reconfiguration mechanisms have not been proposed, nor any concurrency control techniques described in [Cmelik88].

The available process scheme discussed in Section 5.6.1 provides a complete solution to these problems. Conversations can be exploited for reducing the frequency of the distributed consensus protocol. In particular, once a replicated process has agreed to participate in a conversation with a client, request messages can be served without further agreement until the end of the conversation.

A technique resembling the primary copy scheme has been described by Borg et al. [Borg83]. Whenever a message is sent to a process, the same message is forwarded to the backup process. The system ensures that both the processes cannot continue running until it has been verified that both have correctly received the message. Thus, if one process crashes because of any hardware fault, the other one can continue. Furthermore, the remaining process can then clone itself, making a new backup to maintain the fault-tolerance capability.

One disadvantage of Borg's approach is that, if processes exchange messages at a high rate, a considerable amount of CPU time may go into keeping the processes synchronized at each exchange. This disadvantage can be mitigated by adopting the scheme discussed by Powell and Presotto [Powell83]. The system described in that paper puts almost no additional load on the processes being backed up. All messages sent on the network are recorded by a special recorder process. From time to time, each process checkpoints itself onto a remote disk. If a process crashes, recovery is carried out by sending the most recent checkpoint to an idle process which starts running. The recorder process then sends to the newly created process all the messages that the original process received between the checkpoint and the crash. The primary process scheme developed here to work in conjunction with conversations reduces the need for frequent message exchanges by employing the dual of the deferred update technique used in the primary copy scheme.

5.8. Concluding remarks

After examining the structure of a variety of systems, two canonical architectures of fault-tolerant systems were developed, one of which is representative of the techniques and terminology used within the database systems community, the other of which is more closely allied to the real time and process control applications area. These architectures were shown to be duals of each other. Although, in retrospect, this may not appear to be a surprising conclusion, particularly given the Lauer and Needham paper, it has not been realized before how direct and complete the relationship between the two architectures was, and there is not any earlier literature explaining and exploiting this duality. Instead, one finds that fault-tolerant systems are constructed and described using the concepts and terminology applicable to just one of the two architectures, with no apparent realization of the potential relevance of systems and the literature describing them which make use of the other architecture. However, it has to be recognized that the duality previously discussed is sometimes obscured by the fact that many process control applications are structured as a small and fixed number of processes, whereas it is more usual to find object-based systems which contain a large and dynamically varying number of objects.

The arguments to support the duality claim were based on an examination of three properties of a fault-tolerant computation, namely: freedom from interference, backward recovery capability and crash resistance. It was shown that mechanisms employed to implement a given property in one architecture have duals in the other. Similarly, any particular behaviour observed in one architecture has its dual in the other. Examples presented in this chapter show that programs developed using the primitives of one architecture can be mapped easily to the programs of the other architecture. Indeed, it could be claimed that the differences between the two architectures are principally a matter of viewpoint and terminology.

The establishment of the equivalence between the two approaches to fault tolerance has several interesting implications, some of which are enumerated here.

1. There seems to be no inherent reason with respect to fault tolerance for favouring one approach over the other. For example, there is no obvious reason why a real time system must be designed using the primitives of the PM architecture. In fact, one is led to state that the choice for a given system should not be dictated by the application area but by the architectural features of the layer over which the system is to be built.

2. It can also be stated that a single system based on either architecture can in principle, support both classes of applications.

3. It may further be speculated that, were sufficient representative systems of each kind available for detailed evaluation and comparison, it would be found that the observation made in [Lauer78] regarding the invariance of operating system performance under two classes of systems also applies to this fault-tolerance duality.

4. Techniques and mechanisms which happen to have been developed within the domain of just one of the architectures can be mapped and applied to the other architecture. Several examples were presented to illustrate this observation. It was shown that optimization techniques developed for read operations of actions can be applied to optimize conversations. A second example indicated that the exception handling framework developed for the PM architecture can be applied to the OM architecture. Finally, by making use of the duality mapping proposed, interesting techniques for replicated process management were developed. Some existing process replication techniques were also described to show that they are special cases of the schemes derived here.

5. The ideas from this chapter can be used for the design of fault-tolerant systems with minimum set of compatible concepts, thus allowing several degrees of freedom in the design process to be eliminated, leading to well structured systems.

6. Finally, given that, as discussed in [Dobson86], there is the prospect of using certain kinds of fault-tolerance techniques to provide increased security and not just increased reliability, it appears that the duality

mapping presented here can be extended and applied to clarify and illuminate at least some of the literature discussing various approaches to building multi-level secure systems.

Another important issue in distributed systems, whether they are built as object-based or process-based, is that of garbage collection. The following chapter will address specifically the notion of distributed garbage collection, and for convenience the terminology used will be object-based.

Chapter 6

An Example of Object-Based Distributed System Design: Fault-Tolerant Garbage Collection

The function of a garbage collector in a computer system is to reclaim storage that is not needed any more. Developing a garbage collector for a distributed system composed of autonomous computers (nodes) connected by a communication network poses a challenging problem: optimizing performance whilst achieving fault tolerance. This chapter presents the design and implementation of a reference-count garbage collection scheme which is both efficient and fault-tolerant. A distributed object-based system is considered where there can be inter-node object references, and operations on remote objects are invoked via remote procedure calls. The orphan treatment scheme associated with remote procedure calls has been enhanced to enable the collection of garbage arising from node crashes.

First, this chapter contains a brief review of existing work on distributed garbage collection. Next, it gives an overview of an RPC mechanism with an orphan detection and killing facility designed and built at Newcastle. Finally, the enhancements necessary will be described. This chapter is a revised and extended version of work reported earlier [Mancini88b].

6.1. Notes on garbage collection

The function of a garbage collection scheme is to automatically reclaim storage that is no longer in use by computations. This automatic collection of storage frees the programmer from dealing with the complexity of dynamically determining which objects are needed and which are not at any particular time. Storage for objects is allocated from a heap. In simple systems the heap is kept in the primary store, so objects are volatile. An object is defined to be accessible if it is reachable from a fixed object called the root.

The two main garbage collection schemes are (1) mark-scan, and (2) reference-count.

1. A great majority of garbage collectors for non-distributed systems employ the mark-scan technique [Knuth72]. Mark-scan garbage collection needs to be invoked only when there is no free storage available; otherwise it imposes no performance penalty. When the collector is invoked, all other computations are stopped and storage for objects that are not accessible is collected for reuse. Starting from the root, the first phase (mark) causes all references to be traced and every object actually in use to be marked. The scan phase examines the mark on every object; unmarked objects are free and their storage spaces are collected together for reuse.

   A major objection to the mark-scan technique is that all of the ongoing computations must be halted when the collector is invoked. This has the effect of making an application suddenly unresponsive while the collection is taking place. Such unpredictable and often lengthy interruptions are unacceptable in real time applications. In a distributed system the problem is even more serious since work on all nodes must be halted for a global search to take place when any one processor runs out of memory. Another disadvantage is that all objects must be scanned (no matter how many are free), so the cost of this technique is proportional to the total number of objects in the system.

A number of proposals have been made to circumvent these problems. Although versions of mark-scan have been developed which operate in parallel with normal processing [Dijkstra78], the garbage collection is still global in the sense that the entire system needs to be searched.

2. A simple way to automatically collect unused storage is to associate with each object a reference-count field recording the number of references to that object. The reference-count is incremented each time a new reference is created by an object and decremented each time an old reference is removed by an object. When the count falls to zero, no references remain and the storage block can be deallocated [Cohen81].

   Reference counting, unlike mark-scan, does not require that application processes be halted during collection. The overhead due to the algorithm is spread across object manipulations, which makes this technique suitable for real-time and interactive programming environments, as shown in [Eckart87]. Moreover, reference counting is localized, an object can be collected without examining the state of the whole storage. So this technique appears to be suitable for implementing garbage collection in a distributed system. The major objection raised to this scheme is that it cannot collect cyclic linked data structures. An unused cyclic list will not be reclaimed - each individual cell in the list will have a non zero reference-count, although the list as a whole is no longer needed. Several algorithms to solve this problem whilst retaining most of the advantages of reference-count over mark-scan garbage collection have been proposed [Bobrow80, Brownbridge85, Vestal87].

Garbage collection of a single storage heap has been widely discussed for many years; this chapter is concerned with garbage collection in distributed, unreliable systems. In such systems, 'the heap' turns out to be distributed among the nodes of the system. Such a distributed heap can be viewed as a heap whose root is distributed and consists of the union of the roots at all nodes. In such an environment, an object is accessible if it is accessible from one of the roots. Several algorithms to perform distributed garbage collection have been published recently [Hudak82, Ali84, Wiseman89].

Hudak's collection scheme is based on performing a global mark-scan collection beginning at a unique, system-wide root object [Hudak82]. Each object, beginning with the root, first checks if it has been marked. If not, it marks itself, sends a mark message to each object that it references, and awaits replies from all these objects. This may be viewed as each object containing a mark procedure that recursively calls the mark procedures of all objects reachable from it. The collection terminates when the root procedure returns.

Ali describes a number of algorithms for use in a distributed system [Ali84]. The most advanced of his algorithms adopts a technique similar to [Baker78], and does not require any sort of synchronized global collection - a collector only examines a portion of the total space each time. This technique also permits the collector to perform in parallel with other processes. However, his algorithm cannot collect cycles that span more than one node.

Another method has been proposed in [Wiseman89]. Here, a mark-scan algorithm is presented to collect a recursively structured heap, which is partitioned into disjoint (logical) areas. The areas may themselves be partitioned further into more areas, which are collected in parallel exploiting the traditional technique of divide and conquer - the mark-scan of an area is effected by combining the results of the lower level mark-scans rather than with extra phases. If the various areas are located at separated processors of a distributed systems, an additional phase to detect the distributed termination of the mark-scan process is required to prevent accessible objects being marked inaccessible. This method permits the collection of all inaccessible objects, and in particular of those forming circular structures.

None of the methods discussed so far have addressed the problem of fault-tolerant garbage collection in distributed systems. This topic, although important, has not received much attention. The author is only aware of two papers [Liskov86, Vestal87] which address this issue.

The scheme presented in [Liskov86] exploits a reliable central service to store information about inter-node references. The nodes communicate with the central service periodically, to inform it about their references to objects at other sites, and to inquire about the accessibility of any local objects that might be referred to at other sites. Having a central service which deals with inter-node references reduces the problem of distributed garbage collection to a local one, hence allowing the use of standard garbage collection techniques. This approach requires the central service to use a large amount of storage to record the map of the distributed heap - in the worst case such a storage might be as large as the whole distributed heap.

In [Vestal87], two fault-tolerant garbage collection algorithms for object-based distributed systems are presented. The first algorithm combines reference-count with an algorithm to collect circular object structures. Vestal's solution maintains a separate reference-count, called local reference-count, in every node that contains any references for a given object. The object itself contains a list of nodes that have local reference-counts for it. An object obtains the actual reference-count by summing all the local reference-counts. These local reference-counts will continually experience creation, change, and deletion during the operation of the system. The problem then arises of computing a single global reference-count for an object in parallel with other processes. A solution is proposed requiring the synchronization of the physical clocks and the execution of certain procedures atomically with respect to failures. The failure atomicity property is also exploited to guarantee reliable copy of a remote reference among nodes. The second of Vestal's algorithm uses a parallel mark-scan collector based on the algorithm presented in [Dijkstra78]. It resembles the solution in [Ali84], but can collect cycles that span more than one node with high probability.

The scheme presented by Liskov and Ladin is different from the one presented in this chapter in that it employs a centralized (replicated) service for recording object references whereas the latter does not employ such a service. The first of Vestal's solutions has the drawback that to collect a cycle the algorithm needs to start at an inaccessible object lying within the cycle. Finding an effective heuristic for choosing such an object is not simple, and requires research into the exact behaviour of the particular system. The second of Vestal's solutions does not guarantee the collection of inaccessible cyclic structures. This is because it is possible, though quite unlikely, that cyclic structures of inaccessible objects will be moved round a ring of nodes, each node attempting to localize the garbage by passing it on to the next. Vestal does suggest possible ways of reducing the probability of such an event occurring, but these effectively cause the garbage collectors at different nodes to synchronize, which nullifies the benefits of independent garbage collection of nodes.

The next section briefly introduces the model of computation and the terminology employed in the rest of the chapter.

6.2. Object-based garbage collection and reliability requirements

In order to present the garbage collection scheme, a typical implementation of object-based systems is considered. In such an implementation, each object is associated with a unique name - a capability - which is used to control access to the object. A capability is context-independent in that, regardless of where the capability is stored in the system, it always refers to the same object. To emphasize the distributed nature of the system, capabilities for remote objects are referred to as remote capabilities (RCs). The existence of some method is assumed for locating objects efficiently, given these objects' RCs. In such a capability system, an object is treated as garbage, if no capabilities for it exist.

In a distributed object-based system, an operation on a remote object is typically performed by invoking the operation of the object via an RPC with the RC for the object as one of the arguments. Below, the terminology employed is introduced, and illustrated with the help of Figure 6.1, which shows an object x at node B holding an RC for an object y at node A (this is indicated by the dashed line). The node where an object is located is called the owner of the object; the object is local to that node. An object will be termed public if its owner has sent its capability to some other node (so y is a public object). A local object that is not public will be termed private. Some mechanism is required to allow that RCs for remote objects appear the same abstraction as local object capabilities. One such mechanism is illustrated in Figure 6.1, where object x holds an RC for remote object y. Each node maintains two objects called the *export list* and the *import list*. The export

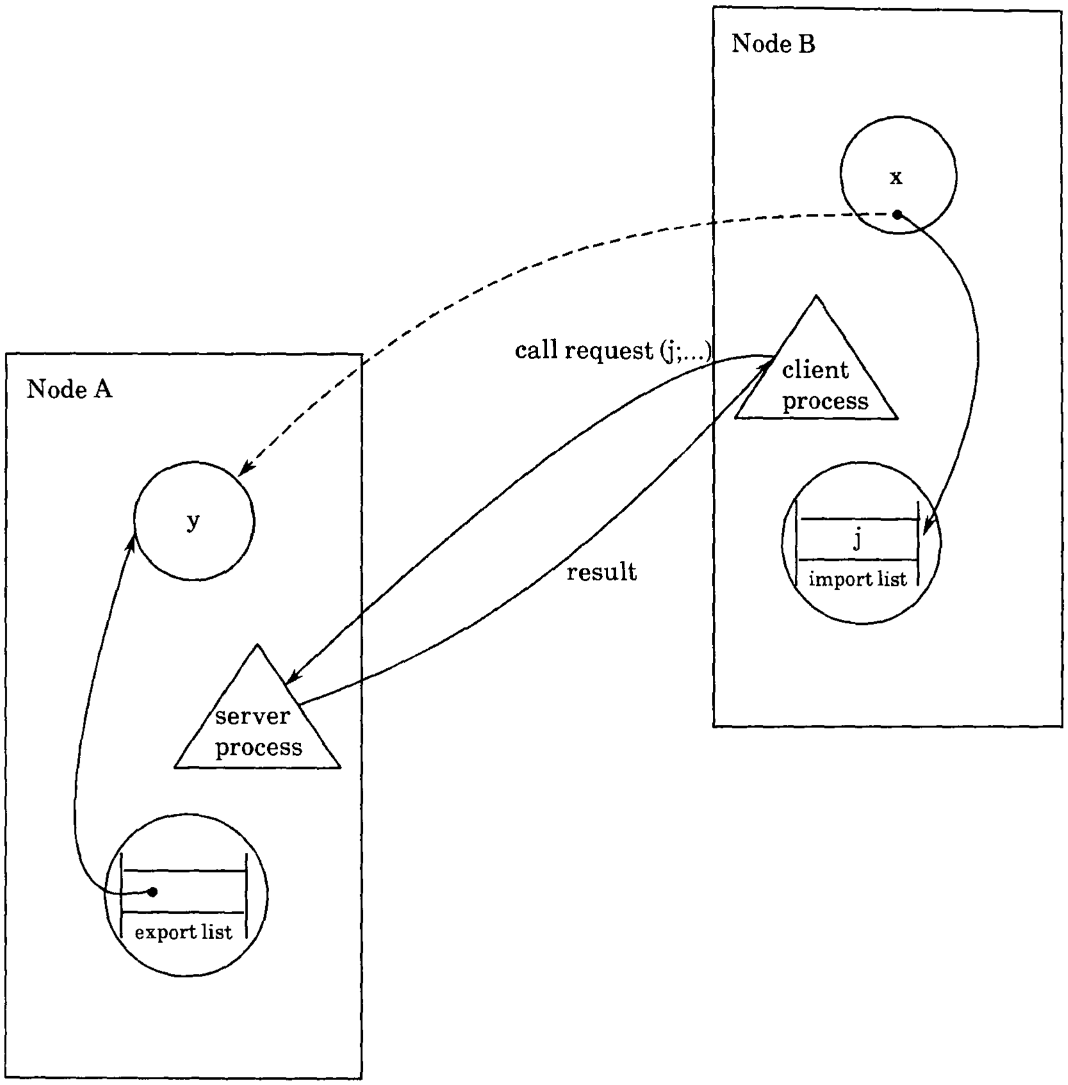

Figure 6.1: Object x holds an RC for object y.

list of a node maintains a list of all public objects of that node, whilst the import list maintains all the RCs of that node. Specific details of how objects come to hold RCs for other objects are not directly relevant for the discussions, so will be glossed over. It will, however, be assumed that objects are capable of transferring (copying) their RCs to other objects.

A distributed computation is performed by client and server processes. The invocation of an operation on y by x will be carried out as follows: a client process at node B obtains RC j for y from the local import list and sends a call request containing j to a server process at A. The server process at A uses the RC j received to get the address of object y from the export list; it then performs the requested operation on y and sends the results back to the client. It will be assumed that a server process can be used for serving a sequence of calls from a given client. Servers and clients are created by the RPC mechanism as the need arises.

It will be assumed that a crash of a node causes volatile objects to be destroyed; in addition a crash also destroys all the processes of that node. A node can also own stable objects which are not destroyed by a crash. There are thus three possible kinds of object-based systems from the point of view of fault tolerance:

1. All objects are volatile (temporary) and are lost with crashes. In such a system, if node B crashes, then x, the client process, the export and import lists of B and therefore the RC for y at node B vanish and the server on A will become an orphan computation. Assuming that only x

held an RC for y, then y will become garbage. It will be assumed that lifetimes of volatile objects do not exceed that of computations which created them; thus, a volatile public object will always have one or more server processes associated with it.

2. All objects are stable (persistent): objects, including import and export lists, survive crashes. The lifetime of stable objects can exceed the lifetime of the computations which manipulate them. In this case, x, the import list and therefore the RC for y survive a crash of B. It is worth noting that such a crash will cause the server process to become an orphan computation, but y will not become garbage. A distributed system with stable objects will typically need to structure its computations as atomic transactions [Gray78] in order to maintain consistency. However, such a provision is orthogonal to the garbage collection scheme.

3. A subset of the objects is stable and the remaining part is volatile. Naturally, only the volatile objects of a node will vanish because of a crash with the possibility of creating garbage on other nodes.

An RC will be called stable if it is held by a stable object, and volatile if it is held by a volatile object. A crash of a node may cause some remote public objects to become garbage. Consider the system shown in Figure 6.2. Suppose that x deletes its RC for y at node A and then a crash of node C occurs; in this case, y becomes garbage and must be reclaimed by the garbage collection system. There is also a second case where node crashes may cause dangling references. For example, a crash of node D in Figure 6.2 will

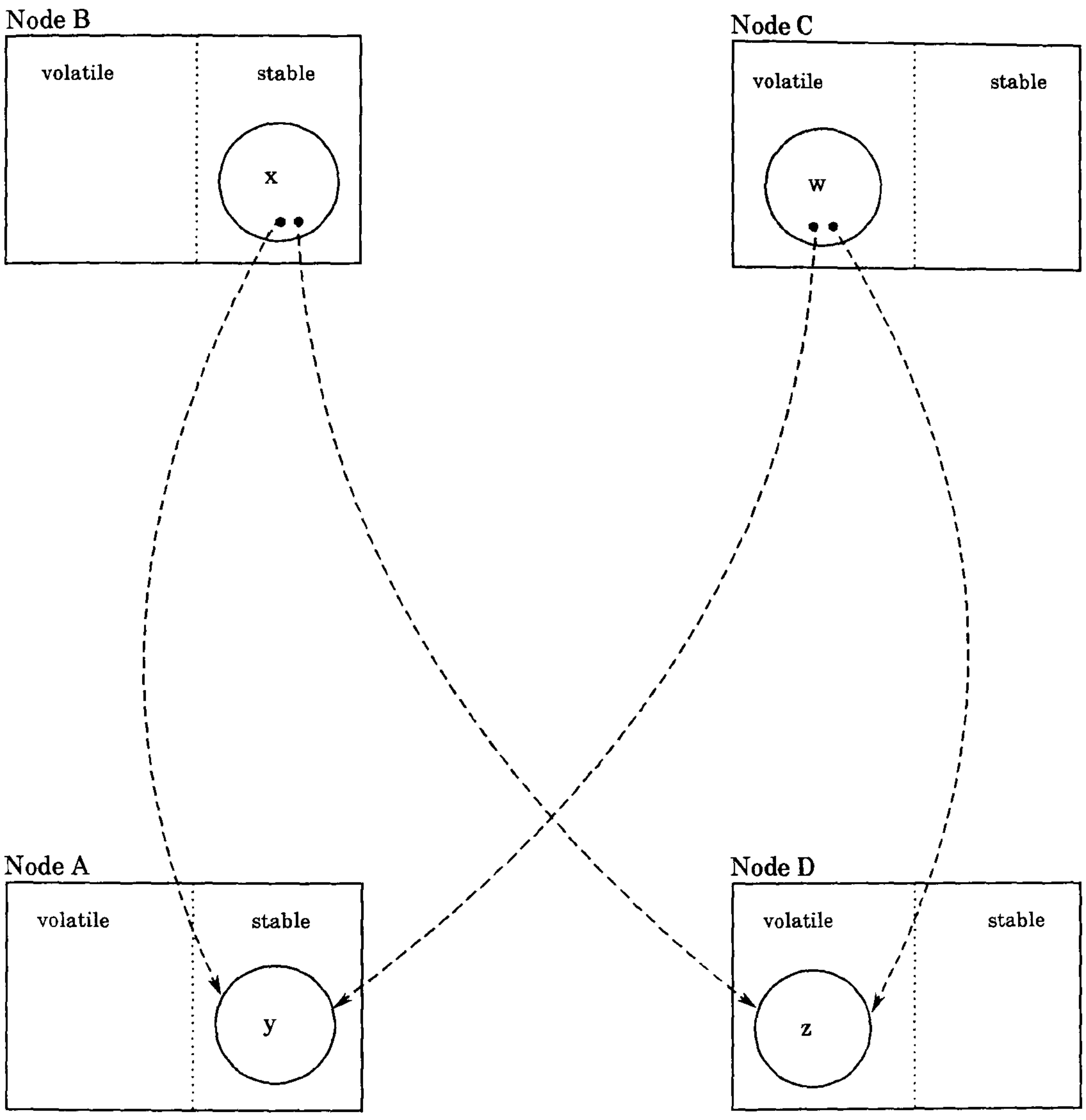

Figure 6.2: Stable and volatile objects and RCs.

cause x and w to hold RCs for z which no longer exists. As such, an invocation of some operation on z by x or w may well cause a run-time exception whose treatment should be orthogonal to the functioning of the garbage collection system. Such dangling references can be detected simply by recording the time at which a node is initialized. This time is made part of all RCs to objects stored in the node. When an RC is used, a check is made to see if the time in the RC is the same as the time when the node was initialized. If not, the node must have crashed and so the RC is invalidated. The following sections will concentrate in particular on developing a distributed garbage collector capable of dealing with the first undesirable situation.

The requirements that a distributed garbage collector for an object-based system should meet are given below.

- The collection scheme should be capable of handling both volatile and stable objects of varying size.

- The scheme should be applicable to both real-time and interactive programming environments. For example, a scheme which required stopping all ongoing computations in the entire system while performing garbage collection would be unacceptable.

- The scheme should be fault-tolerant to failures that occur during collection. In a distributed system, part of the system can fail while other parts still function. This behaviour imposes various reliability-related requirements on garbage collection. For example, collection of garbage created by node crashes should be guaranteed at non-crashed nodes, and the collection mechanism should be able to cope with the transfer of RCs among nodes in the presence of failures.

- In most distributed systems, sending a message from one process to some remote process is a relatively slow operation (consuming anything from a few to several milliseconds of time), so the garbage collection scheme should strive to minimize network communication requirements.

The distributed garbage collection scheme presented here has several interesting features: (1) it is tolerant to the following types of failures: node crashes (fail-silent behaviour will be assumed, that is, a crashed node completely ceases to function), and lost, duplicated, delayed and out-of-order messages; (2) it does not require elaborate facilities such as failure-atomic procedures or synchronized clocks; (3) individual nodes in the system are free to choose any local garbage collection technique; (4) the design can rely on a close integration with the orphan detection scheme of a remote procedure call mechanism, thus enabling the exploitation of existing fault-tolerance facilities.

In the scheme presented in this chapter, relevant information about inter-node references is stored at each node using a technique based on the reference-count method. There are two correctness requirements for a reference counting garbage collector:

SAF: if the reference-count of an object is zero, then there are no references for that object;

LIV: if there are n (n $\geq$ 0) references in the system for an object, then that object will eventually have its reference-count equal to n.

Bearing in mind that in a reference counting scheme an object is collected if and only if its reference-count is zero, these two requirements can be seen as the statements of safety and liveness properties. The first requirement, SAF, states the safety property that nothing bad happens (viz. referred objects do not get collected), but it does not ensure that something good happens: the garbage collector might leave all objects with positive reference-counts (viz. never collect any objects) and still satisfy SAF. The liveness property LIV is therefore needed to guarantee that actual progress does take place. The liveness property requires the updating of reference-counts.

In the next section, the orphan detection method that will be used as a basis for the fault-tolerant reference-count service will be briefly presented.

6.3. RPCs and orphan detection and killing

Orphans are unwanted executions that can manifest themselves due to communication or node failures [Nelson81]. In the following it is assumed an exactly-once semantics for RPCs: a normal termination (the client receives a reply from the called server) implies exactly one execution. An abnormal termination can mean zero, partial or one execution at the called server. A call is said to terminate abnormally if the termination occurs because no reply message is received from the called server. Network protocols typically employ timeouts to prevent a process waiting for a message from being held up indefinitely. Based on the client-server model, Panzieri and Shrivastava [Panzieri88] have recently developed an efficient technique for orphan treatment for RPCs with exactly-once semantics. There are three mechanisms used for treating orphans:

(i) Every call contains a *deadline,* indicating to the server the maximum time available for execution. If the deadline expires, then the server aborts the execution and the call terminates abnormally. It is worthwhile to note that if there are no node crashes in the system, then this mechanism will be enough to cope with orphans. The remaining two mechanisms cope with crashes.

(ii) Every node maintains a variable - called the *crashcount* - which is initialized to the current value of the local stable clock immediately after a node recovers from a crash. A node also maintains a table of crashcount values for clients that have made calls to it. A call request contains the client's crashcount value - if this value is greater than the one stored in the table at the called server node, then there could be orphans at the server node which are first aborted before proceeding with the call.

(iii) Every node has a *terminator process* that occasionally checks the crashcount values of other nodes - by sending messages to them and receiving replies from those that are up - and aborts any orphans when it detects any crashes.

These mechanisms have been optimized to provide a cheap orphan treatment system. In particular, no stable storage is required (other than the stable clock which is available in most computers anyway) and there is no need to keep clocks synchronized. Further, the terminator based mechanism has been optimized as follows: a server that has not received calls from a client for a while marks itself as a potential orphan. The terminator need only perform its checks for potential orphans. Finally, the RPC mechanism copes with message failures (lost, duplicated and delayed messages) by employing well-known protocol-related techniques which will not be discussed here.

Given that each node has an orphan detection facility, it seems natural to embellish it for garbage detection. Referring to the example discussed previously, a crash of a node B can leave garbage at node A, which can be detected by node A while detecting orphans. Such an integrated orphan detection and garbage collection mechanism is the main subject of the subsequent sections. In particular, what follows will describe enhancements made to the mechanisms (ii) and (iii) above to provide garbage collection.

6.4. Fault-tolerant garbage collection

The main features of a distributed fault-tolerant garbage collection scheme exploiting the above orphan treatment system will be presented in the following sections. Section 6.4.1 presents a simple fault-tolerant scheme for volatile objects to be used when transferring of RCs is not permitted. In Section 6.4.2 the refinements required to cope with RC transfers will be discussed. Only the mechanisms (ii) and (iii) of the orphan detection scheme discussed in the previous section will be exploited, namely every node is required to maintain a crashcount and to run a terminator process occasionally. Note that mechanism (i) is not considered, because reclaiming garbage in the presence of node and communication failures is the central topic of this scheme. Section 6.4.3 discusses how the scheme can be extended to cope with stable objects. Since the scheme presented here is based on the reference-count technique, it suffers from the well-known limitation that it is incapable of collecting objects if inter-node references form a cycle. However, it will be shown in Section 6.4.4 that this limitation can be removed by extending the design.

6.4.1. Treatment of node failures

Nodes are responsible for doing local garbage collection. Only private objects are candidates for garbage collection at a node. Each local garbage collector treats the objects not accessible from the local root as garbage. Since the export list is always accessible from the local root, all the public objects not accessible through the export list become private. Therefore the problem of designing a fault-tolerant distributed garbage collector reduces to the design of a protocol to keep the exports lists consistent with the import lists throughout the distributed system. Note that public objects may be used locally as well; these objects will be collected only when neither local nor remote capabilities exist for them.

At each node there is a reference-count service, integrated into the RPC mechanism, which is responsible for determining the accessibility of public objects. The reference-count service of a node achieves its aims by updating the export and import lists mentioned earlier. An entry is added to the export list the first time a capability for a local object is sent to another node (i.e. when a private object becomes public). This entry includes a reference-count field indicating the number of objects that hold RCs for this public object. The export list provides the local garbage collector with the information necessary for detecting objects that are no longer public (an object whose reference-count field in the export list reaches zero becomes private and therefore a candidate for garbage collection if no local references exist).

The objects listed in the export list may be a superset of those actually used by other nodes. For example, referring to Figure 6.1, suppose that x at node B holds the only RC for y at node A, and that x is deleted at B. Object y is no longer accessible, yet there will be a positive reference-count in A's export list until some further action is taken at A.

During local garbage collection, the collector is required to construct a *junk list* of all the imported RCs deleted, and then update the import list after finishing the local garbage collection. The reference-count service of a node is responsible for distributing the junk list to other nodes. Thus the receiving nodes are provided with the information necessary to update their export list in order to assess the accessibility of their public objects. The junk list need not be kept stable because the garbage due to node crashes is detected by the orphan detection mechanism. Each node does its garbage collection independently of other nodes, using an algorithm of its choice. The algorithm must however be extended slightly to take account of the export, import and junk lists. It is worth noting that the construction of the junk list can be performed without any additional scan of the storage.

A data structure referred to as *client list,* which is a list of ClientElem records (see Figure 6.3), is maintained by the RPC orphan detection scheme at a node and contains information about all the client nodes that have made calls to this particular node [Panzieri88]. An entry of type RClist is required for garbage detection purposes.

```
type ClientElem = struct (  Name clientNode   % client node address %
                            Real crashCount   % crash count value of the clientNode %

                            % list of servers created for the clientNode %
                            ServerList serverList

                            % list of the public objects used by the clientNode %
                            RClist rcList  );
```

Figure 6.3: Client list data structure.

The RClist lists the public objects capabilities that have been used by the client whose name is recorded in the clientNode. The serverList field contains the names of local servers which have been created for the clientNode. The client list and export list of a node are initialized to be empty at the node startup time.

The protocol followed at each node in order to support the distributed garbage collection service will now be discussed.

When a capability for a local object at some node A is to be exported to some other node B as a result of a call request invoked by a client process at node B, the called server process running on node A performs the following steps:

(1) If the export list at node A contains an entry for the capability being exported, then its reference-count value is incremented by one, otherwise a new element is added to the export list, with the reference-count field initialized to one;

(2) The capability being exported is inserted in the rcList field of the entry for node B in the client list at node A.

Whenever an orphan server is aborted at node A because a crash of node B is detected, either by some server at node A or by the terminator of node A (respectively mechanism (ii) and (iii), Section 6.3), the following steps are performed at node A:

(1) All the public objects recorded in the rcList field of the client list entry for node B get their reference-count values in the export list at node A decremented by one. Entries with reference-count field containing zeros are deleted from the export list thus making the relevant objects private;

(2) The entry for node B is removed from the client list. Thus ensuring that the previous step is performed only once.

A node, say B, periodically sends its junk list to other nodes. Upon receiving this list every node performs the following operations for each RC in the junk list sent by node B:

(1) It checks if the RCs sent by B correspond to any public object in the rcList field of the entry for node B in the client list, and if so,

(2) It deletes the relevant public object capability from rcList and decrements the appropriate reference-count field of the export list by one. If the field is zero then that entry is deleted as stated earlier.

It is worth noting that inaccessible cyclic structures of RCs can be collected if a crash of a node breaks the cycle. In this case orphan servers of that node will be detected on at least one other node forming the cycle thus causing the storage for the cyclic structure to be reclaimed.

The above mentioned operations represent minor modifications to the existing orphan detection and killing system whose design has been analyzed and shown correct in a formal setting in [Pappalardo88]. It is worth noting that the scheme presented so far ensures SAF and LIV requirements in the absence of RC transfers. SAF, which requires that only those objects for which no RCs exist (viz. those objects with reference-count equal to zero) become private, is ensured because the objects listed in the export list are always a superset of those actually needed by other nodes. The reference-count of a public object, say y, is decremented only after either (1) some node holding an RC for y crashed causing the RC to vanish, or (2) some node sent a junk list containing the RC for y. LIV is ensured in the presence of crashes because orphan servers will eventually be aborted, thus causing the updating of the relevant reference-counts.

Inconsistencies can arise due to crash of nodes during the transfer of RCs. In this case the scheme presented so far does not ensure that only objects without RCs for them will have reference-counts equal to zero. In the following section this and other issues will be discussed.

6.4.2. Reliable transfer of remote capabilities

One additional mechanism is required to transfer RCs reliably while preserving SAF and LIV. Consider the following example. Node A is the owner of a public object and node B holds an RC for that object. B now transfers this RC to some node C as a result of a request by C. Inconsistencies can arise if B crashes (causing its RC to vanish) after sending its RC to C, but before informing A about the RC transfer. In this case SAF may be violated - because the public object owned by A can be garbage collected, leaving C to hold an RC for a non-existing object. In order to satisfy SAF, the RC transfer should only be regarded as completed normally if the export list of A and its client list have been updated properly. Consider then the following protocol. Whenever a server discovers that it is transferring an RC as a part of its RPC reply message, it first informs (see the inform message in Figure 6.4, where numbers indicate the sequence in which the messages are sent) the owner of the relevant object so that the owner can update the export list and make an entry in the client list (for C in this

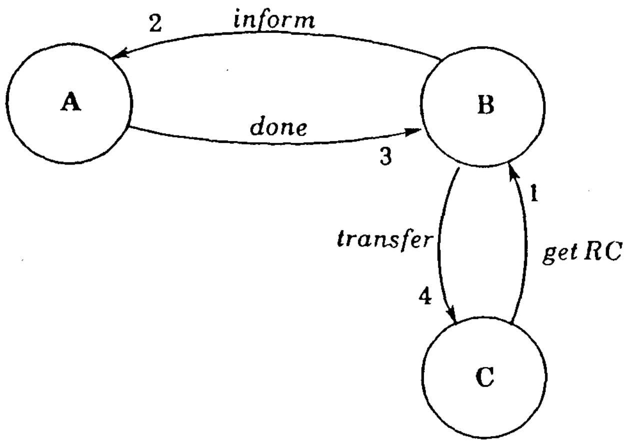

Figure 6.4: An RC transfer.

case). Only after receiving the done message does the server send the reply message transfer with the RC. Referring to the example, if the call by C to acquire the RC from B terminates normally, it is ensured that the export list and client list at A have been updated. Thus the protocol guarantees the SAF requirement.

Now consider situations where the LIV requirement can be violated. With reference to Figure 6.4, suppose that B crashes after informing A but before sending the transfer message to C. In this case LIV may be violated - the object reference-count in A may be higher than the number of RCs in the system. The terminator and potential orphan mechanism mentioned in the previous section can be suitably modified to cope with such situations. The potential orphan mechanism operating at A makes sure that if no calls are received from B or C for a long time, then enquiry messages will be sent to them to detect crashes. This mechanism can be enhanced to take care of *unused objects,* that is public objects that are listed in the client list, and that remain unused for a long time. In the situation presented above, eventually A will send an enquiry message to C and will be able to adjust its relevant entries since C does not holds the RC. It should be noted that the protocol discussed can be seen as a technique for A to cope with crashes of B; crashes of C are dealt with by A in the same manner.

To summarize, nodes are periodically required to exchange three types of information: (i) lists of potential orphans, (ii) lists of unused objects, and (iii) junk lists. The first type of information is required for orphan detection, and the remaining two for garbage detection. A simple optimization is for a node to construct a single message containing all the three components for distribution.

6.4.3. Treatment of stable objects

The scheme presented so far deals with the treatment of volatile objects. This section will discuss enhancements of two kinds: to cope with stable objects (i.e. objects are persistent and survive crashes), and to cope with the mixed approach where both stable and volatile objects are permitted.

In order to implement the abstraction of a stable heap, all the bookkeeping information about stable objects must also be kept stable, therefore each node must maintain its export, import, and junk lists on stable storage. Since these data structures are kept stable, node crashes cannot produce garbage on other nodes. The protocols discussed in Section 6.4.1 need only one modification: the updating of the export list when orphan servers are detected is no longer required - the export list of a node is updated only when a junk list is received. However, the mechanism discussed in Section 6.4.2 for preserving SAF and LIV is still required for transferring RCs between nodes, as the following example illustrates. Suppose the transfer protocol is not employed, then the following situation is possible (Figure 6.4): B deletes its RC after sending it to C and then crashes before informing A about the transfer. If garbage collection is done at A using post-crash information from B (note that the junk lists are kept stable while the information about the RC transfers are not), the object referred by the RC at C might be collected by mistake. An alternative to our solution for solving the above possible inconsistency is to keep also a stable log of all in-transit references [Liskov86].

Now consider the provision of garbage collection in distributed systems where both volatile and stable objects are supported. In such a system volatile and stable RCs for the same object are permitted (e.g. RCs to y in Figure 6.2). An example of such an environment could be a network of nodes some of which are diskless workstations. In such a system RCs held by diskless workstations are volatile and if such workstations crash garbage might be created in other nodes.

In order to implement such a mixed scheme, it is necessary to record the type of RCs a node holds; this can be performed in the client list (see Figure 6.3) by requiring each element of the rcList in the client list to contain two fields - the RC offered to the client node, and in addition a flag indicating whether the RC is stable or not. The bookkeeping information regarding objects (export, and import lists, and the junk list) can also be split in two parts with lists on volatile store recording information about volatile objects and stable lists recording information about stable objects. Naturally, a public object will become private only when its reference-count becomes zero on both the export lists. Given this organization, the garbage collection schemes presented for volatile and stable objects can coexist - whenever orphan servers are aborted at a node, reference-counts of only those RCs which are recorded as volatile in the client list are decremented in the volatile export list. This mixed approach continues to satisfy both SAF and LIV properties. For example, if C crashes (refer to Figure 6.2) then the reference-count of y will be decremented by one when that crash is detected at A; however, y will not be deleted because there still exists a stable RC naming y at B.

The scheme presented here has similar functionality to that given in [Liskov86] with the following differences: (i) there is no need to keep in-transit references on stable storage; any inconsistencies caused by crashes during an RC transfer are detected and removed by the enhanced orphan detection scheme discussed; (ii) the scheme provides a uniform way of treating both volatile and stable objects.

6.4.4. Inter-node cycles

If inter-node references form an acyclic graph, then, when an object of that acyclic graph is collected, all the garbage objects reachable from that object will eventually be deleted. However, if some inaccessible inter-node references form a cycle, inaccessible objects will never be deleted in the scheme proposed so far, as indeed in any pure reference counting scheme. For example, suppose object x at node A has a reference to object y at node B and y has a reference to object x, as shown in Figure 6.5.

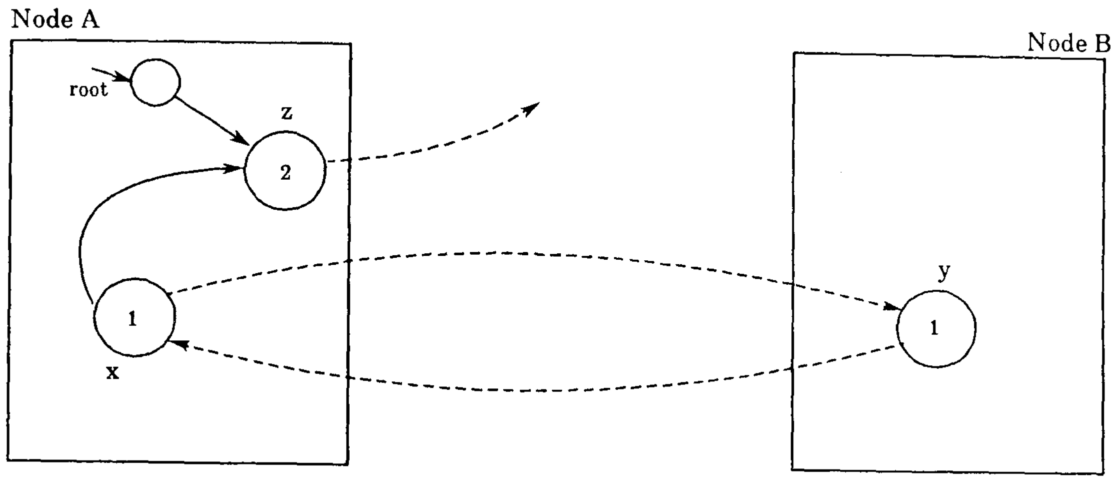

Figure 6.5: An inter-node cycle.

The inter-node references for x and y form a cycle that spans node boundaries. Even though x and y are both locally inaccessible, they appear to be globally accessible and therefore are not reclaimed by the local garbage collector at their nodes. They are also not recognized as inaccessible with the scheme presented in the preceding sections. In the following, two possible approaches will be shown to remove this limitation by extending the design with the ideas reported respectively in [Wiseman89] and in [Vestal87, Brownbridge85].

One way to devise a cycle-tracing scheme is to employ, for example, the distributed mark-scan garbage collection proposed by Wiseman [Wiseman89].

In order to perform a global marking of the storage, each node in the Wiseman scheme helds a list which records the marking information for the public objects which are remotely referenced by other nodes. The mark field can take the values: *not found, found,* and *scanned.* Initially all objects are marked as *not found.* When an object is first found to be accessible its mark is changed to *found.* Once all the references reachable from the object marked as *found* have been examined, the object is marked as *scanned.* The global marking phase terminates once no *found* objects remain. At this point, all the objects which are marked as *scanned* must be kept, but the objects marked as *not found* are known to be inaccessible and so can be collected. This collection takes place during the final scanning phase, which is local to the nodes. It should be noted that the objects marked as *not found* include those forming inter-node cycles, for example object x and y in Figure 6.5 will still be marked as *not found* at the end of the global marking phase.

In order to employ the Wiseman scheme to achieve a fault-tolerant cycle-tracing algorithm, the problem of detecting the termination of the marking phase must be solved. Consider the situation depicted in Figure 6.6, where the marking information has been included in the entry of the export list at each node. In such a situation, if node A crashes, the remote objects y and w at node B will remain marked as *not found* until A recovers. During such time, B could wrongly collect object w. In general, B cannot know whether an object marked as *not found* is still needed. For example, although objects w and y are marked as *not found,* object w is still accessible, while object y is part of an inaccessible inter-node cycle.

The solution chosen for this problem is to mark as *found* all the objects being referred to by a crashed node. As previously discussed, the scheme presented in this chapter discovers when a node crashes. To complete the marking phase each node should lookup its local client list (see Section 6.4.1) and mark as *found* all the objects used by the crashed nodes. In the example of Figure 6.7, when node A crashes, objects w and y will be marked as *found*. After the marking phase terminates, the scanning phase will collect the inter-node cycles through all non-faulty nodes.

This distributed cycle-tracing scheme, therefore, will collect only part of the inaccessible inter-node cycles, and there is the question of whether this can cause the entire system to stop, because of shortage of storage. For example, with reference to Figure 6.7, in the case that only node A crashes, the inter-node cycle spanning node B and C will be collected, that is objects o and q, while the inter-node cycle between A and B will not be collected, at least while A remains crashed. However, it is worth noting that the number of inaccessible inter-node cycles cannot increase. A crashed node which prevents the collection of a cycle also prevents the creation of an additional cycle, because since the node is crashed another cycle cannot be made through it.

The cycle-tracing algorithm presented above can be optimized when inter-node cycles include volatile objects. For example, if object x at node A in Figure 6.6 is volatile, and A crashes, then object y at node B can be collected without waiting until the end of the cycle-tracing algorithm. This speeds up termination of the cycle-tracing algorithm and allows early

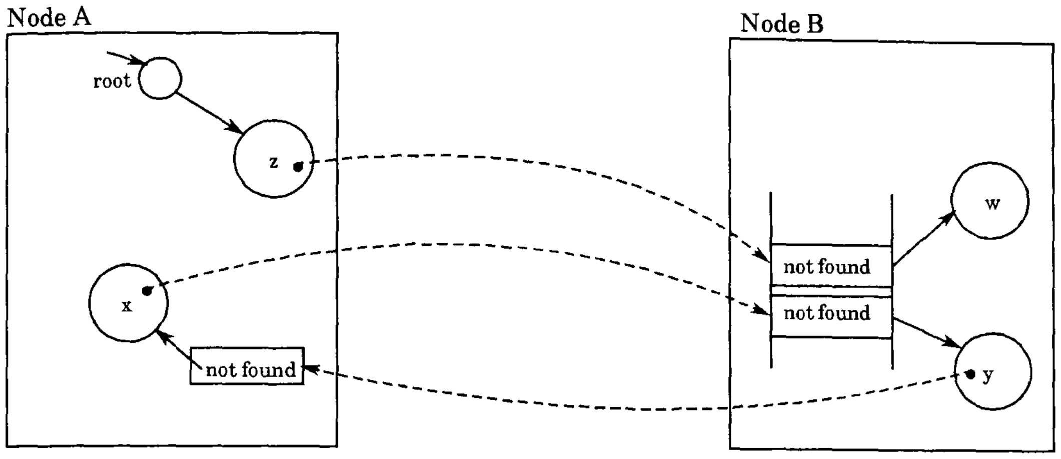

Figure 6.6: Cycle tracing initialization.

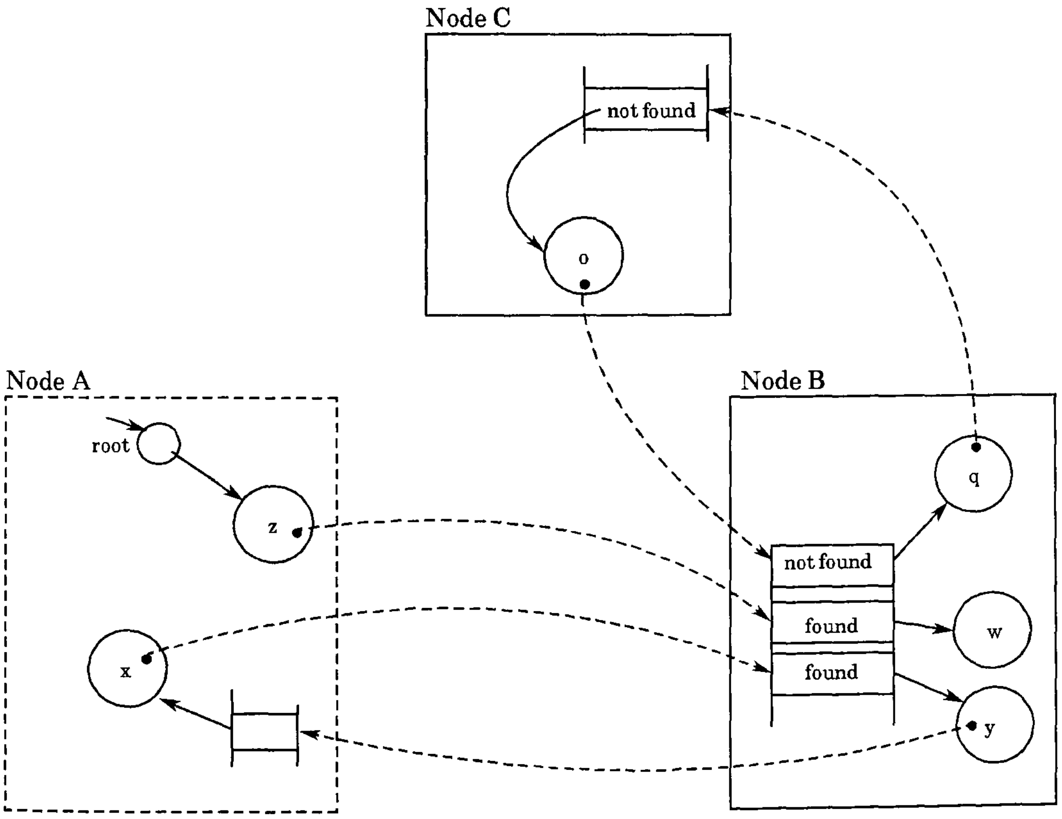

Figure 6.7: Marking after node A crashes.

collection of volatile objects forming inter-node cycles spanning through crashed nodes.

An alternative approach to a global mark-scan algorithm, is to perform a local scan for deleting inaccessible objects when it is believed that a cycle has formed. Various algorithms of this kind have been proposed in the literature [Vestal87, Brownbridge85]. For example, Vestal proposes an algorithm that, when started at an inaccessible object lying in a cycle, will collect the entire cycle. This algorithm requires a suitable heuristic for selecting the starting object, otherwise the collection of the cycle cannot be guaranteed. Vestal's algorithm is simple and can be easily integrated in the reference counting scheme discussed previously. The only problem with Vestal's solution is to provide an effective heuristic to detect possible cycles. This can be achieved in the following way.

As discussed previously, every node has a terminator process such that when an object remains unused for a long time that process sends enquiry messages to the clients to make sure they are still running. The terminator process can be suitably enhanced to provide an effective heuristic for Vestal's algorithm. If an unused object, say x at node A in Figure 6.6, is lying in a cycle, then a message, sent by the terminator process of node A to enquire about the remote clients of object x, will come back to node A after propagation through the nodes in the cycle. Therefore object x can be chosen to start executing Vestal's algorithm. It should be noted that the terminator process, on receiving back its enquiry message for object x, can only deduce that x lies in a cycle, but not that x is an inaccessible object.

Both the schemes discussed above require a crashed node to recover in order to collect inter-node cycles going through that node. This should not be a cause of any increase in the number of cycles, because the number of cycles though a crashed node remains fixed during the down-time of that node. However, in order to assess which scheme performs better, a further analysis is required to establish the relationships between the rate of collection and the rate of production of inter-node cycles. Developing an appropriate statistical model is a demanding task, because of the large number of parameters that need estimation. Those involve the number of collectors, the relative priorities between collectors and users processes, as well as patterns of usage like locality of references.

6.4.5. Performances

A prototype version of the basic design presented has been implemented on a network of Flex object-based systems [Foster82] running on ICL Perq workstations connected by an Ethernet. The implemented prototype did not need to cater for cycles of inaccessible objects, because these cannot be formed in the Flex system. To the extent it could be tested, the distributed garbage collection scheme worked as specified, in particular it collected garbage objects in the presence of node crashes and communication failures.

Measurements of the overhead caused by the scheme have been made. The measurements were made on a lightly loaded Ethernet. The Ethernet had a raw data rate of 10 megabits per second and was shared with other users.

The measurements have been carried out for procedure calls performed by a client process to a remote existing server, which returns a new RC at each call. The average time taken for such a call to complete was measured. This time interval includes the time spent by the client looking up the import list and inserting in the import list the newly created RC when the call returns, plus the time the server spends updating the export list and client list. The average time per call (averaged over 1000 calls) was 51 milliseconds, while the same call without any provisions for orphan-detection and garbage collection took 44 milliseconds on average. The performance degradation due to this scheme is thus of the order of 16%.

By running various distributed programs, it was also noted that, of the volatile garbage created in the entire distributed system, less then 1% was global. This result is consistent with the experiments reported for a similar system in [Wiseman89]. Therefore it is expected that the memory utilization should not change sensibly because of node failures. In particular, a negligible variation was noted in the percentage of garbage collected by the local garbage collectors in a system without failures, and a system where node crashes where caused.

The influence of the rate for the distribution of the junk lists on performance was not measured. By empirical observations, it appears that the collection of public objects could be done at intervals of the order of several minutes without affecting the overall performance of the system. The local garbage collection of every node runs at a much higher rate and is capable of providing the required storage for ongoing computations.

6.5. Concluding remarks

The topic of fault-tolerant garbage collection in distributed systems, although important, has not received much attention. A practical solution has been presented in this chapter.

The distributed garbage collection discussed here handles fault tolerance by an extension of orphan killing techniques, and this in fact is yet another illustration of the duality between process-based and object-based architectures. Because orphans are essentially garbage processes, it is perhaps not surprising that the orphan detection schemes can be extended to deal with garbage objects. However, somehow embarrassingly, the author has to admit that this facet of the duality argument was not as readily clear at the outset of the development as now is.

The garbage collection scheme presented involves small modifications to an efficient orphan treatment scheme implemented at Newcastle [Panzieri88], so there is every reason to believe that the technique is of practical value. The performance figures presented bear out this observation. Some of the advantages of the distributed garbage collection scheme presented here are given below:

1. Collection takes place asynchronously with respect to other activities, including local garbage collection, and creation and deletion of private and public objects;
2. It is independent of the local garbage collection schemes employed at various nodes;

3. It is tolerant to node crashes and communication failures that occur during collection;
4. It is capable of treating both volatile and stable objects;
5. It does not require elaborate facilities such as failure-atomic procedures or synchronized clocks.

The scheme as described here has been developed for an object-based architecture with RPC. However, there is no reason why the reference-count service could not be implemented on its own for process-based architectures, though process-based architectures are usually tailored for much smaller numbers of objects, whose garbage collection is not likely to be as serious a problem as in a typical object-based architecture.

Chapter 7

Conclusions

This thesis has concentrated on the development of techniques for reliable distributed architectures, and has argued that work on object-based architectures can also have relevance to more conventional architectures. This final chapter summarizes the material that has been presented and indicates some of the possible areas for future research.

7.1. Summary of the thesis

The current literature ascribes many merits to the object-based programming methodology, see for example [Dahl70, Birtwhistle73, Goldberg83, Almes85]. However, there is a considerable confusion in the literature about exactly what the terms 'object-based', and 'object-oriented' mean. For example, MacLennan employs the term 'object-oriented' to give a definition of Computer Science ([MacLennan82], p. 75):

> *"It might not be unreasonable to call computer science objectified mathematics, or object-oriented mathematics"*

while Pascoe begins one of his recent papers saying [Pascoe86]:

> *"There are as many different views of what object-oriented programming is as there are computer scientists and programmers"*

It is difficult to give a succinct definition, except at a gross level, since there are many subtle flavours of behaviour which combine to give the overall picture. Therefore, this thesis started by giving, in Chapter 2, a precise characterization of the relevant features of the object-based programming methodology. Based on their dependence relations, four approaches have been identified. One of these approaches is the object-based methodology where objects, that is encapsulation of some data together with the set of operations on that data, are grouped into classes through which the concept of abstract data types is provided. The classes themselves can be organized into a class hierarchy. Such hierarchies allow similar classes to be related together in such a way that the code implementing the behaviour of one class can be automatically re-used (inherited) by classes lower in the hierarchy, thus simplifying the implementation of those lower-level classes.

Most of the literature regards object-based programming as though it can only be done with special programming languages. One of the avenues that has been explored in the present research has been the use of object-based programming techniques in a procedure-based language. This work, that has been described in Chapter 3, arose from particular work on garbage collection in distributed systems, which has also been reported in Chapter 6.

In Chapter 3, two general techniques have been presented for implementing the sub-classing form of inheritance as a set of extensions to a procedure-based language. Both techniques exploit first-class procedures. The first technique requires some run-time lookup, while the second technique establishes all the bindings at the time objects are created. These two solutions provide different characteristics and advantages. If one is essentially interested in quick prototyping and experimentation, the first approach may be better adapted. Keeping the dispatching scheme during execution makes it easier to change the class hierarchy on the spot in order to correct errors, improve the system, or experiment with new facilities since it is not necessary to go through a complete compilation of other classes in the hierarchy on each occasion. On the other hand, if one is concerned about correctness, robustness and efficiency, then the second approach, which constructs most of the bindings at compile-time, is obviously required.

The subsequent chapters of this thesis have therefore considered object-based programming, whether this is done with an explicit language or by means of the techniques described in Chapter 3. In particular, Chapter 4 has reviewed some applications of the object-based methodology to the design and implementation of distributed architectures.

Distributed architectures are characterized by the physical partitioning of their components. This partitioning, which requires explicit communications between different physical components, introduces a number of fundamental issues concerning the visibility of distribution. Various forms of distribution transparency have been identified in [ANSA87]. They can be regarded both as problems to be solved in order to conceal the partitioning of architectural components, and as features to be exploited to take advantage of the partitioning in order to achieve particular levels of security, reliability, and performance. Chapter 4 has started from the analysis in [ANSA87] to discuss various strategies, abstractions, and mechanisms required for controlling and exploiting distribution in object-based architectures.

Just as Chapter 3 has illustrated a means of applying object-based methodologies without using an explicit object-based programming language, Chapter 5 has discussed work which illustrates the relationship between distributed object-based architectures and an apparently different form of distributed architectures, based on processes. After examining the structure of a variety of systems, two canonical architectures of fault-tolerant systems were developed, one encompassing the techniques and terminology used within the database and office information systems community, the other being more closely allied to the real time and process control applications area. These architectures were shown to be duals of each other.

Although, in retrospect, this may not appear to be a surprising conclusion, particularly given the Lauer and Needham paper, it has not been realized before how direct and complete the relationship between the two architectures was, and there is not any earlier literature explaining and exploiting this duality. Instead, one finds that fault-tolerant systems are constructed and described using the concepts and terminology applicable to just one of the two architectures, with no apparent realization of how useful the methodologies devised for one approach could prove for the other.

The arguments to support the duality claim were based on an examination of three properties of a fault-tolerant computation, namely: freedom from interference, backward recovery capability and crash resistance. It was shown that mechanisms employed to implement a given property in one architecture have duals in the other. Similarly, any particular behaviour observed in one architecture has its dual in the other. Examples presented in Chapter 5 show that programs developed using the primitives of one architecture can be mapped easily to the programs of the other architecture. Indeed, it could be claimed that the differences between the two architectures are principally a matter of view point and terminology. The establishment of the equivalence between the two approaches to fault tolerance has several interesting implications, some of which have also been analyzed in Chapter 5.

Another important issue in distributed systems, whether they are built as object-based or process-based, is that of garbage collection. The topic of fault-tolerant garbage collection in distributed systems has not received much attention. Chapter 6 has addressed to the notion of distributed garbage collection, specifically from the point of view of fault tolerance computing, and has presented a practical solution which is both cheap and efficient.

The distributed garbage collection discussed in Chapter 6 is tolerant to node crashes and communication failures that occur during collection, is capable of treating both volatile and stable objects, and is asynchronous with respect to other activities, including local garbage collection, and creation and deletion of private and public objects.

It should be noted that the proposed garbage collection scheme handles fault tolerance by extending the techniques commonly employed for killing orphan processes, and this is yet a further illustration of the duality between process-based and object-based architectures. Because orphans are essentially garbage processes, it is perhaps not surprising that orphan detection schemes can be extended to deal with garbage objects.

7.2. Future work

The design of distributed architectures offers considerable scope for further investigation. Some particular areas of the thesis which could benefit from further research will now be considered.

In Chapter 3, two techniques for sub-classing where developed for the Flex system [Foster82]. Unfortunately, the Flex environment does not provide a suitable interface for exploiting the full benefit of such techniques. One of the major problems that programmers face in Flex to the difficulty of finding reusable software components, once such components have been produced. Another related problem is how a user can understand the structure of an application, especially if that user was not the creator of that software.

In order to provide a more convenient use of object-based techniques in the Flex system, the question arises of what kinds of tools might help to solve these problems. Many researchers and developers are already addressing these issues in the context of object-based architectures, where the answer is typically couched in terms of finding classes or operations that have specific properties or functions, and of understanding the class hierarchy. Since the Flex environment is mainly procedure-based, one could think of developing the work presented in Chapter 3, in order to adopt the tools designed within the context of object-based architectures.

The work at DEC on an environment for the Trellis/Owl object-based language [O'Brien87] appears to be relevant for this purpose. The Trellis environment addresses the problem of how to find relevant software components when the programmer is not familiar with the entire system, and also keeps track of static inconsistencies and does incremental recompilation automatically.

Chapter 5 has discussed the duality between object-based and process-based systems from an essentially empirical point of view. An interesting area for further research would be that of providing a more formal treatment of this duality. This would, of course, improve the duality arguments, but would also shed light on formal methods for specification and verification of object-based systems. In fact, the insights gained with the duality argument could help in building the dual model of some well-established formalism for studying process-based systems, such as the one presented in [Hoare85]. Such a dual model could be used as the basis of an appropriate formal theory for object-based systems.

One of the results of the research described in this thesis has been the development and implementation of a fault-tolerant garbage collector for distributed systems. In order to improve the performance of the present implementation, another area of future work might concern specific measurements about how such systems will actually be used. Some of the measurements that would be interesting to evaluate, in particular with respect to distributed object-based systems, are: the percentage of references on a node to objects on another node, the rate at which references are transferred from one node to another, and the percentage of objects that are never referred on another node throughout their entire lifetime. Such measurements will of course be strongly influenced by the applications and the usage profile of the system. However, given such measurements, there are various possible trade-off decisions which would be evaluated more carefully, such as the relative priorities between collectors and users processes.

An interesting work would be validating the distributed garbage collection scheme presented in Chapter 6. This requires research into the area of formal specification and verification for real-time programs, since for most applications a basic rate of collection must be met.

References

Abelson85.

Abelson, H. and Sussman, G.J., *Structure and interpretation of computer programs,* The MIT Press/McGraw-Hill Book Company, Cambridge, Massachusetts, 1985.

Agha86.

Agha, G., *Actors,* MIT Press, 1986.

Ali84.

Ali, K.A.H.M., "Object-oriented storage management and garbage collection in distributed processing systems," PhD Thesis, Royal Institute of Technology, Stockholm, Sweden, December 1984.

Almes85.

Almes, G.T., Black, A.P., Lazowska, E.D., and Noe, J.D., "The Eden system: a technical review," *IEEE Trans. on Software Engineering*, vol. 11, no. 1, pp. 43-59, January 1985.

Alsberg76.

Alsberg, P.A. and Day, J.D., "A principle for resilient sharing of distributed resources," *Proceedings of the 2nd International Conference on Software Engineering,* pp. 562-570, IEEE, San Francisco, October 1976.

Anderson81a.

Anderson, T. and Lee, P.A., *Fault Tolerance: principles and practice,* Prentice-Hall, 1981.

Anderson81b.

Anderson, T. and Knight, J. C., "Practical software fault tolerance for real-time systems," Technical Report TR169, Computing Laboratory, University of Newcastle upon Tyne, August 1981.

ANSA87.

ANSA, "The ANSA reference manual," ed. A.J. Herbert and J. Monk, June 1987.

Atkinson87.

Atkinson, M.P. and Buneman, O.P., "Types and persistence in database programming languages," *ACM Computing Surveys*, vol. 19, no. 2, pp. 105-190, June 1987.

Avizienis84.

Avizienis, A. and Kelly, J.K.J., "Fault tolerance by design diversity: concepts and experiments," *IEEE Computer*, vol. 17, no. 8, pp. 67-80, August 1984.

Baker78.

Baker, H.G., "List-processing in real time on a serial computer," *Comm. ACM*, vol. 21, no. 4, pp. 280-294, April 1978.

Banatre78.

Banatre, J.-P. and Shrivastava, S.K., "Reliable resource allocation between unreliable processes," *IEEE Transactions on Software Engineering*, vol. SE-4, no. 3, pp. 230-241, May 1978.

Bernstein84.

Bernstein, P.A. and Goodman, N., "An algorithm for concurrency control and recovery in replicated distributed databases," *ACM Trans. on Database Systems*, vol. 9, no. 4, pp. 596-615, December 1984.

Bernstein87.

Bernstein, P.A., Hadzilacos, V., and Goodman, N., *Concurrency control and recovery in database systems,* Addison-Wesley, Reading, Massachussetts, 1987.

Best81.

Best, E. and Randell, B., "A formal model of atomicity in asynchronous systems," *Acta Informatica,* vol. 16, pp. 93-124, 1981.

Birman85.

Birman, K.P., Joseph, T.J., Raeuchle, T., and Abbadi, A. El, "Implementing fault-tolerant distributed objects," *IEEE Transactions on Software Engineering,* vol. SE-11, no. 6, pp. 502-508, June 1985.

Birrell84.

Birrell, A.D. and Nelson, B.J., "Implementing remote procedure calls," *ACM Transactions on Computer Systems,* vol. 2, no. 1, pp. 39-59, February 1984.

Birtwhistle73.

Birtwhistle, G. M., Dahl, O-J., Myhrhaug, B., and Nygaard, K., *Simula begin,* Academic Press, 1973.

Black86.

Black, A., Hutchinson, N., Jul, E., and Levy, H., "Object structure in the Emerald system," *ACM SIGPLAN Notices (OOPSLA '86 Proceedings),* vol. 21, no. 11, pp. 78-86, November 1986.

Bobrow80.

Bobrow, D.G., "Managing reentrant structures using reference counts," *ACM Transactions on Programming Languages and Systems*, vol. 2, no. 3, pp. 269-273, July 1980.

Bobrow87.

Bobrow, D.G. *et al.*, "Common Lisp Object System specification," ANSI X3J13 Document 87-002, American National Standards Institute, Washington, DC, 1987.

Borg83.

Borg, A., Baumback, J., and Glazer, S., "A message system supporting fault tolerance," *ACM Operating Systems Review*, vol. 17, no. 5, pp. 90-99, October 1983.

Borning81.

Borning, A.H., "The programming language aspects of ThingLab: a constraint-oriented simulation laboratory," *ACM Trans. on Programming Languages and Systems*, vol. 3, no. 4, pp. 353-387, October 1981.

Borning82.

Borning, A.H. and Ingalls, D.H., "Multiple inheritance in Smalltalk 80," *Proc. of the National Conference in Artificial Intelligence*, American Association of Artificial Intelligence, Pittsburgh, Pennsylvania, August 1982.

Borning86.

Borning, A.H., "Classes versus prototypes in object-oriented languages," *ACM/IEEE Fall Joint Computer Conference*, pp. 36-40, Dallas, TX, November 1986.

Brownbridge85.

Brownbridge, D.R., "Cyclic reference counting for combinator machines," *Lecture Notes in Computer Science 201*, pp. 273-288, Springer Verlag, Berlin, September 1985.

Campbell86.

Campbell, R.H. and Randell, B., "Error recovery in asynchronous systems," *IEEE Trans. on Software Engineering*, vol. SE-12, no. 8, pp. 811-826, August 1986.

Cardelli85.

Cardelli, L. and Wegner, P., "On understanding types, data abstraction, and polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, pp. 471-522, December 1985.

Cmelik88.

Cmelik, R., Gehani, N.H., and Roome, W.D., "Fault-tolerant concurrent C: a tool for writing fault-tolerant distributed programs," *18th Int. Symposium on Fault-Tolerant Computing,*, pp. 56-61, Tokyo, Japan, June 1988.

Cohen81.

Cohen, J., "Garbage collection of linked data structures," *ACM Computing Surveys*, vol. 13, no. 3, pp. 341-367, September 1981.

Cook88.

Cook, S., "Varieties of inheritance," *ACM SIGPLAN Notices (OOPSLA '87 Proceedings Addendum)*, vol. 23, no. 5, pp. 35-40, May 1988.

Cox86.

Cox, B.J., *Object-oriented programming - an evolutionary approach,* Addison-Wesley, Reading, Massachusetts, 1986.

Cristian82.

Cristian, F., "Exception handling and software fault tolerance," *IEEE Transactions On Computers*, vol. C-31, no. 6, pp. 531-540, June 1982.

Curry84.

Curry, G.A. and Ayers, R.M., "Experience with Traits in the Xerox Star workstation," *IEEE Trans. on Software Engineering*, vol. SE-10, no. 5, pp. 519-527, September 1984.

Dahl70.

Dahl, O-J., Myhrhaug, B., and Nygaard, K., "Common base language," Technical Report S-22, Norwegian Computing Center, Oslo, Norway, October 1970.

Dahl72.

Dahl, O-J., Dijkstra, E.W., and Hoare, C.A.R., *Structured programming,* Academic Press, 1972.

Davies73.

Davies, C.T., "Recovery semantics for a DB/DC system," *Proceedings of the ACM National Conference*, pp. 136-141, Atlanta, Georgia, August 1973.

Dijkstra72.

Dijkstra, E.W., "The humble programmer," *Comm. ACM*, vol. 15, no. 10, pp. 859-866, October 1972.

Dijkstra78.

Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S., and Steffens, E.F.M., "On the fly garbage collection," *Communications of the ACM*, vol. 21, no. 11, pp. 966-975, November 1978.

Dixon87.

Dixon, G.D. and Shrivastava, S.K., "Exploiting type-inheritance facilities to implement recoverability in object-based systems," *Proc. of the 6th Symposium on Reliability in Distributed Software and Database Systems*, pp. 107-114, Williamsburg, March 1987.

Dobson86.

Dobson, J.E. and Randell, B., "Building reliable secure systems out of unreliable insecure components," in *Proceedings Conference on Security and Privacy*, IEEE, Oakland, April 1986.

DoD80.

DoD, *Reference manual for the ADA programming language*, 1980.

Eckart87.

Eckart, J.D. and LeBlanc, R.J., "Distributed garbage collection," *ACM SIGPLAN Notices*, vol. 22, no. 7, pp. 264-273, July 1987.

Eswaren76.

Eswaren, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L., "The notions of consistency and predicate locks in a database system," *Communications of the ACM*, vol. 19, no. 11, pp. 624-633, November 1976.

Foster82.

Foster, J.M., Currie, I.F., and Edwards, P.W., "Flex: a working computer with an architecture based on procedure values," *Proc. of International Workshop on High-Level Architecture*, pp. 181-185, Fort Lauderdale, Florida, December 1982.

Fowler85.

Fowler, R.J., "Decentralized object finding using forwarding addresses," PhD Thesis 85-12-1, Dept. of Computer Science, University of Washington, Seattle, December 1985.

Goldberg83.

Goldberg, A. and Robson, D., *Smalltalk-80 The Language and its Implementation,* Addison-Wesley, 1983.

Gray78.

Gray, J.N., "Notes on data base operating systems," in *Operating Systems: An Advanced Course*, ed. R. Bayer, R.M. Graham and G. Seegmueller, pp. 393-481, Springer, New York, 1978. (Lecture Notes in Computer Science 60)

Harland84.

Harland, D.M., *Polymorphic programming languages: design and implementation,* John Wiley & Sons, Chichester, U.K., 1984.

Hendler86.

Hendler, J. and Wegner, P., "Viewing object-oriented programming as an enhancement of data abstraction methodology," *Proc. of the Hawaii Conference on System Sciences,* January 1986.

Hoare72.

Hoare, C.A.R., "Proof of correctness of data representations," *Acta*

Informatica, vol. 1, no. 4, pp. 271-281, 1972.

Hoare85.

Hoare, C.A.R., *Communicating sequential processes,* Prentice-Hall, London, 1985.

Horning76.

Horning, J.J., "Some desirable properties of data abstraction facilities," *ACM SIGPLAN Notices*, vol. 11, Special Issue, pp. 60-62, 1976.

Hudak82.

Hudak, P. and Keller, R.M., "Garbage collection and task deletion in distributed applicative processing systems," *Proc. ACM Symposium on Lisp and Functional Languages*, pp. 168-178, Pittsburgh, Pennsylvania, August 1982.

Jacky87.

Jacky, J.P. and Kalet, I.J., "An object-oriented programming discipline for Standard Pascal," *Comm. ACM*, vol. 30, no. 9, pp. 772-776, September 1987.

Jones78.

Jones, A.K., "The Object Model: a conceptual tool for structuring software," in *Lecture Notes in Computer Science 60,* ed. R. Bayer, R.M. Graham and G. Seegmueller, pp. 8-16, Springer, Berlin, 1978.

Kernighan78.

Kernighan, B. W. and Ritchie, D. M., *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

Knuth72.

Knuth, D.E., *The art of computer programming, vol. 1: fundamental algorithms,* Addison-Wesley, Reading, Massachusetts, 1972.

Koo87.

Koo, R. and Toueg, S., "Checkpointing and rollback recovery for distributed systems," *IEEE Trans. on Software Engineering,* vol. SE-13, no. 1, pp. 23-31, January 1987.

Lauer78.

Lauer, H.C. and Needham, R.M., "On the duality of operating system structures," *Proceedings 2nd International Symposium on Operating Systems,* IRIA, October 1978. (Reprinted in Operating Systems Review, vol. 13, no. 2, pp. 3-19, April 1979.)

Lieberman86.

Lieberman, H., "Using prototypical objects to implement shared behaviour in object-oriented systems," *ACM SIGPLAN Notices (OOPSLA '87 Proceedings),* vol. 21, no. 11, pp. 214-223, November 1986.

Liskov77.

Liskov, B., Snyder, A., Atkinson, R., and Schaffert, C., "Abstraction mechanisms in CLU," *Comm. ACM,* vol. 20, no. 8, pp. 564-576, August 1977.

Liskov81.

Liskov, B. *et al.,* "CLU reference manual," *Lecture Notes in Computer Science 114,* Springer Verlag, Berlin, 1981.

Liskov82.

Liskov, B. and Scheifler, R., "Guardians and actions: linguistic support for robust, distributed programs," *Proceedings of the 9th Annual Symposium on Principles of Programming Languages*, pp. 7-19, Albuquerque, New Mexico, January 1982.

Liskov86.

Liskov, B. and Ladin, R., "Highly available distributed services and fault-tolerant distributed garbage collection," *Proc. 5th ACM Symposium on Principles of Distributed Computing*, pp. 29-39, Calgary, Alberta, Canada, August 1986.

Liskov87.

Liskov, B. *et al.*, "Implementation of Argus," *Proc. 11th ACM Symposium on Operating Systems Principles*, pp. 111-122, Austin, Texas, November 1987.

Liskov88.

Liskov, B., "Data abstraction and hierarchy," *ACM SIGPLAN Notices (OOPSLA '87 Proceedings Addendum)*, vol. 23, no. 5, pp. 17-34, May 1988.

MacLennan82.

MacLennan, B.J., "Values and objects in programming languages," *ACM SIGPLAN Notices*, pp. 70-79, December 1982.

Mancini86a.

Mancini, L.V., "Modular redundancy in a message passing system," *IEEE Trans. Software Engineering*, vol. SE-12, no. 1, pp. 79-86, January 1986. (Also Technical Report TR209, Computing Laboratory, University of Newcastle upon Tyne, November 1985)

Mancini86b.

Mancini, L.V. and Shrivastava, S.K., "Exception handling in replicated systems with voting," *16th Int. Conf. on Fault Tolerant Computing*, pp. 384-389, Vienna, Austria, July 1986. (Also Technical Report TR217, Computing Laboratory, University of Newcastle upon Tyne, May 1986)

Mancini87.

Mancini, L.V. and Shrivastava, S.K., "Collecting garbage while detecting orphans in distributed system is both cheap and efficient," System Research Memorandum SRM/452, Computing Laboratory, University of Newcastle upon Tyne, January 1987.

Mancini88a.

Mancini, L.V., "A Technique for subclassing and its implementation exploiting polymorphic procedures," *Software Practice & Experience*, vol. 18, no. 4, pp. 287-300, April 1988.

Mancini88b.

Mancini, L.V. and Shrivastava, S.K., "Fault-tolerant reference counting for garbage collection in distributed systems," Technical Report TR/260, Computing Laboratory, University of Newcastle upon Tyne, June 1988.

Mancini89.

Mancini, L.V. and Shrivastava, S.K., "Object and process replication: a case study in fault tolerance duality," *19th Int. Conf. on Fault Tolerant Computing*, Chicago, Illinois, June 1989. (To appear. Also System

Research Memorandum SRM/471, Computing Laboratory, University of Newcastle upon Tyne, December 1988)

McKendry85.

McKendry, M.S. and Herlihy, M., "Time-driven orphan elimination," CMU-CS-85-138, Dept. of Computer Science, Carnegie-Mellon University, July 1985.

Milner84.

Milner, R., "A proposal for standard ML," *ACM Symposium on Lisp and Functional Programming*, pp. 184-197, 1984.

Mohan83.

Mohan, C. and Lindsay, B., "Efficient commit protocols for the tree of processes model of distributed transactions," *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, pp. 76-88, Montreal, August 1983.

Moon86.

Moon, D.A., "Object-oriented programming with Flavors," *ACM SIGPLAN Notices (OOPSLA '86 Proceedings)*, vol. 21, no. 11, pp. 1-8, November 1986.

Nelson81.

Nelson, B.J., "Remote procedure call," PhD Thesis, CMU-CS-81-119, Dept of Computer Science, Carnegie-Mellon University, Pittsburgh, 1981.

O´Brien87.

O´Brien, P.D., Halbert, D.C., and Kilian, M.F., "The Trellis programming environment," *ACM SIGPLAN Notices (OOPSLA '87 Proceedings)*, vol. 22, no. 12, pp. 91-102, December 1987.

Panzieri85.

Panzieri, F., "Design and development of communication protocols for local area networks," Technical Report TR197, Computing Laboratory, University of Newcastle upon Tyne, March 1985.

Panzieri88.

Panzieri, F. and Shrivastava, S.K., "Rajdoot: a remote procedure call mechanism supporting orphan detection and killing," *IEEE Trans. on Software Engineering*, vol. 14, no. 1, pp. 30-37, January 1988.

Pappalardo88.

Pappalardo, G. and Shrivastava, S.K., "A formal treatment of interference in remote procedure calls," in *Lecture Notes in Computer Science 331*, ed. M. Joseph, pp. 209-227, Springer Verlag, Berlin, 1988.

Parnas72.

Parnas, D.L., "On criteria to be used for decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, p. 1053, December 1972.

Pascoe86.

Pascoe, G.A., "Elements of object-oriented programming," *Byte magazine*, pp. 139-144, August 1986.

Powell83.

Powell, M.L. and Presotto, D.L., "Publishing: a reliable broadcast communication mechanism," *Proceeding 9th ACM Symposium on Operating*

Systems Principles, pp. 100 - 109, October 1983. (Operating Systems Review, vol. 17, no. 5)

Randell64.

Randell, B. and Russell, L.J., *Algol 60 implementation,* Academic Press, New York, 1964.

Randell75.

Randell, B., "System structure for software fault tolerance," *IEEE Transactions on Software Engineering,* vol. SE-1, no. 2, pp. 220-232, June 1975.

Randell78.

Randell, B., Lee, P.A., and Treleaven, P.C., "Reliability issues in computing system design," *ACM Computing Surveys,* vol. 10, no. 2, pp. 123-165, June 1978.

Schaffert86.

Schaffert, C., Cooper, T., Bullis, B., Kilian, M.F., and Wilpolt, C., "An introduction to Trellis/Owl," *ACM SIGPLAN Notices (OOPSLA '86 Proceedings),* vol. 21, no. 11, pp. 8-16, November 1986.

Schneider87.

Schneider, F.D., "The state machine approach: a tutorial," Technical Report 86-800, Dept. of Computer Science, Cornell University, June 1987.

Shrivastava88a.

Shrivastava, S.K., Dixon, G.D., Hedayati, F., Parrington, G.D., and Wheater, S.M., "A technical overview of Arjuna: a system for reliable distributed computing," *IEE U.K. IT 88 Conference*, pp. 601-605, Swansea, Wales, July 1988.

Shrivastava88b.

Shrivastava, S.K., Mancini, L.V., and Randell, B., "On the duality of fault tolerant system structures," in *Lecture Notes in Computer Science 309*, ed. J. Nehmer, pp. 19-37, Springer Verlag, Berlin, 1988.

Sloman87.

Sloman, M. and Kramer, J., *Distributed systems and computer networks*, Prentice-Hall, London, 1987.

Smith88.

Smith, J.M., "A survey of process migration mechanisms," *ACM Operating Systems Review*, vol. 22, no. 3, pp. 28-40, July 1988.

Snyder86a.

Snyder, A., "Encapsulation and inheritance in object-oriented programming languages," *ACM SIGPLAN Notices (OOPSLA '86 Proceedings)*, vol. 21, no. 11, pp. 38-45, November 1986.

Snyder86b.

Snyder, A., "CommonObjects: an overview," *ACM SIGPLAN Notices*, vol. 21, no. 10, pp. 19-28, October 1986.

Stein87.

Stein, L.A., "Delegation is inheritance," *ACM SIGPLAN Notices (OOPSLA '87 Proceedings)*, vol. 22, no. 12, pp. 138-146, December 1987.

Stonebraker79.

Stonebraker, M., "Concurrency control and consistency of multiple copies of data in distributed INGRES," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 3, pp. 188-194, May 1979.

Stroustrup86.

Stroustrup, B., *The C++ programming language,* Addison-Wesley, Reading, Massachusetts, 1986.

Stroustrup88.

Stroustrup, B., "What is object-oriented programming?," *IEEE Software,* vol. 5, no. 3, pp. 10-20, May 1988.

Svobodova84.

Svobodova, L., "Resilient distributed computing," *IEEE Transactions on Software Engineering* , vol. SE-10, no. 3, pp. 257-267, May 1984.

Tanenbaum81.

Tanenbaum, A.S. and Mullender, S.J., "An overview of the Amoeba distributed operating system," *ACM Operating Systems Review*, vol. 15, no. 3, pp. 51-64, July 1981.

Tanenbaum85.

Tanenbaum, A.S. and Renesse, R. Van, "Distributed operating systems," *ACM Computing Surveys*, vol. 17, no. 4, pp. 419-469, December 1985.

Tanenbaum86.

Tanenbaum, A.S. and Mullender, S.J., "The design of a capability-based distributed operating system," *The Computer Journal*, vol. 29, no. 4, pp. 289-300, 1986.

Taylor86.

Taylor, D.J., "Concurrency and forward recovery in atomic actions," *IEEE Trans. on Software Engineering*, vol. SE-12, no. 1, pp. 69-78, January 1986.

Ungar87.

Ungar, D. and Smith, R.B., "Self: the power of simplicity," *ACM SIGPLAN Notices (OOPSLA '87 Proceedings)*, vol. 22, no. 12, pp. 227-242, December 1987.

Vestal87.

Vestal, S.C., "Garbage collection: an exercise in distributed fault-tolerant programming," PhD Thesis, Dept. of Computer Science, University of Washington, Seattle, 1987.

Wegner83.

Wegner, P., "On the unification of data and program abstraction in Ada," *Proc. of the 10th ACM Symposium on Principles of Programming Languages*, pp. 256-264, 1983.

Wegner87a.

Wegner, P., "The object-oriented classification paradigm," in *Research Directions in Object-Oriented Programming*, ed. B. Shriver, P. Wegner, pp. 479-560, MIT Press, 1987.

Wegner87b.

Wegner, P., "Dimensions of object-based language design," *ACM SIGPLAN Notices (OOPSLA '87 Proceedings)*, vol. 22, no. 12, pp. 168-182, December 1987.

Wegner88.

Wegner, P., "Discussion sessions of the workshop on object-oriented programming," *ACM SIGPLAN Notices*, vol. 23, no. 1, pp. 16-37, January 1988.

Wensley78.

Wensley, J. *et al.*, "SIFT: design and analysis of a fault-tolerant computer for aircraft control," *Proc. of the IEEE*, vol. 60, pp. 1240-1254, October 1978.

Wirth83.

Wirth, N., *Programming in Modula-2,* Springer, Second Edition, 1983.

Wiseman89.

Wiseman, S.R., "Garbage collection in distributed systems," PhD Thesis, Computing Laboratory, University of Newcastle upon Tyne, 1989.

Wood81.

Wood, W.G., "A decentralised recovery control protocol," *Digest of Papers FTCS-11: Eleventh Annual International Symposium on Fault-Tolerant Computing,* pp. 159-164, Portland, June 1981.

Woodward83.

Woodward, P.M. and Bond, S.G., *Guide to Algol-68 for users of RS systems,* Edward Arnold Ltd., London, U.K., 1983.

Yonezawa86.

Yonezawa, A. *et al.*, "Object-oriented concurrent programming in ABCL/1," *ACM SIGPLAN Notices (OOPSLA '86 Proceedings),* vol. 21, no. 11, pp. 258-268, November 1986.

Zimmerman80.

Zimmerman, H., "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communicatons*, vol. COM-28, pp. 425-432, April 1980.