Selective Transparency in Distributed Transaction Processing

Daniel L. McCue

Ph.D. Thesis

The University of Newcastle upon Tyne
Computing Laboratory

April 1992

Abstract

Object-oriented programming languages provide a powerful interface for programmers to access the mechanisms necessary for reliable distributed computing. Using inheritance and polymorphism provided by the object model, it is possible to develop a hierarchy of *classes* to capture the semantics and inter-relationships of various levels of functionality required for distributed transaction processing. Using multiple inheritance, application developers can selectively apply transaction properties to suit the requirements of the application objects.

In addition to the specific problems of (distributed) transaction processing in an environment of persistent *objects*, there is a need for a unified framework, or architecture in which to place this system. To be truly effective, not only the transaction manager, but the entire transaction support environment must be described, designed and implemented in terms of objects.

This thesis presents an architecture for reliable distributed processing in which the management of persistence, provision of transaction properties (e.g., concurrency control), and organisation of support services (e.g., RPC) are all gathered into a unified design based on the object model.

Acknowledgements

"Induced, by a conviction of the great utility of such engines, to withdraw for some time my attention from a subject on which it has been engaged during several years, and which possesses charms of a higher order, I have now arrived at a point where success is no longer doubtful." [Charles Babbage 1822]

I sincerely thank my supervisor, Professor Santosh Shrivastava, who has taught me so much about reliability and distributed systems. His unfailing generosity with time, ideas and collaborative work is sincerely appreciated. He has been an inspiration and a role model for the highest level of professionalism.

Professor Pete Lee, a trusted friend and adviser, deserves my thanks for bringing me to Newcastle in the first place and for his diligent reading of early drafts of this thesis. His patience, stamina and encouragement have kept me going and encouraged me to complete the work once started.

I would also like to thank the many members of the Arjuna project, particularly Mark Little, Graham Parrington and Stuart Wheater for their help and support.

The members of the Computing Laboratory, especially Ron Kerr and Graham Megson, have also been a continuing source of inspiration and encouragement, keeping me intellectually challenged and making my stay here all the more pleasant.

The work described in this thesis has been supported in part by grants from the UK Science and Engineering Council and ESPRIT project No. 2267 (Integrated Systems Architecture).

Finally, I must thank my wife, Charlene, who has always been there when I needed her.

Table of Contents

List of Figures

List of Tables

1 Introduction

"Contrary to the situation with hardware, where an increase in reliability usually has to be paid for by a higher price, in the case of software unreliability is the greatest cost factor."
Edsger W. Dijkstra [Dijkstra 1982]

1.1 Developing Reliable Software

As businesses, governments and individuals become increasingly dependent on computer systems, the reliability of computer software and systems becomes ever more important. Software and hardware design techniques that were once employed only for "life–critical" computer applications are now recognised to be necessary for a wide range of "business–critical" functions. Modern approaches to software design *should* show increasing use of techniques to improve reliability, and increasingly reliable software should result.

Unfortunately, some of the otherwise beneficial trends in computer systems development, such as better communications and increased computing power, have had an adverse effect on the level of reliability which has been achieved. Networking technology has advanced to a point to allow even heterogeneous machine interconnection with relative ease. However, reliable management of distributed data remains a difficult, burdensome chore for programmers. Furthermore, the problems of data and code distribution, management of communications and system failures, and access to remote data have been added to the already difficult task of application programming. Larger computer networks and ever faster CPUs further tempt applications designers to attempt ever more complex projects. The effect of the increased size and complexity of computer applications is to reduce their reliability. Thus, while business demand for reliability is increasing, technology trends are eroding the level of reliability that programmers can achieve with present-day tools.

The difficulties of producing reliable software for distributed systems are manifest in many ways. Some of the problems stem directly from the addition of distribution to already complex applications. Other problems arise because distribution introduces more possibilities for failure, which causes overall reliability to suffer. Three fundamental application requirements which are difficult to meet reliably in distributed applications are data sharing, data (and code) distribution and data storage. Each requirement gives rise to problems which programmers must resolve in developing reliable distributed applications.

- **Data Sharing** – As computer systems are applied to a wider variety of applications, business data is increasingly viewed as a resource to be shared – indeed a valuable corporate asset. Maintaining consistency of shared data is an absolute requirement of reliable applications. In distributed systems, as in non-distributed systems, concurrent access to shared data must be controlled to prevent inconsistencies from arising. When a single object is involved, for example in recording a deposit into a single bank account, access may be controlled through the use of a simple locking protocol. Different applications accessing the bank account can follow a protocol of acquiring the lock first, accessing the account, then releasing the lock. Consistency of the account balance will be ensured for this simple case, assuming that all applications obey this protocol when accessing the account.

  In general however, a more sophisticated mechanism is required to maintain consistency and maximise concurrency for simultaneous accesses to multiple objects. For example, when multiple objects using this protocol are involved in a single logical operation, as in a transfer from one bank account to another, no single lock will cover both accounts simultaneously. If locks on the individual accounts are to be acquired in sequence, a protocol must be employed for lock acquisition and release to ensure consistency and avoid deadlock. Expressing appropriate concurrency controls on access to application data adds complexity to the programmer's interface and introduces a potential source of errors in programming. Programmers need a systematic way to control interactions among concurrent uses of data.

- **Distribution of data and programs** – As computers are increasingly linked together using sophisticated networks, it becomes necessary to access both remote and local data at the same time from a single program. Benefits that potentially accompany distribution, such as increased availability, reliability and performance, incur a cost in increased implementation complexity. Not only are the problems of locating and accessing remote data added to the programming problem, but also new kinds of system failures may occur when access to shared data may involve communications with remote machines. Programmers require a reliable means of locating and accessing data which may be distributed throughout a large network of machines.
- **Storage of data** – A major source of complexity in the design and development of computer applications is the management of persistent data. The process of describing persistent application data and data relationships, traditionally associated with database systems, is a science in itself. Countless hours of human effort have been applied to the design and development of modern database systems. Yet, the *programming interfaces* used for creating, manipulating and destroying persistent data are virtually unchanged since the first data processing languages were developed. These traditional interfaces are primitive, requiring each programmer to encode application abstractions into linear, unstructured representations for storage, and then decode these unstructured representations when the stored data is accessed again. This encoding/decoding is tedious and error-prone, reducing overall reliability and increasing the level of complexity which programmers must manage. Improved programming language support for persistent data must be provided.

Research into database systems and programming languages has produced a variety of methods, tools and techniques to address these three requirements. The following sections of this chapter summarise the key concepts of these techniques. However, existing solutions are not well integrated with each other or with modern programming languages. A proliferation of programming interfaces, addressing these problems piecemeal, causes undue complexity for application programmers, inviting errors and reducing overall reliability.

1.2 The Programming Interface

A programming language is the tool with which a programmer expresses the intent of the application. Until constructs for sharing, distribution and storage of data are smoothly integrated into a programming language, the development of reliable distributed applications will remain a monumental chore. To control this proliferation of interfaces and thus simplify the programming effort requires a coherent set of abstractions which render "transparent" the various mechanisms for sharing, distribution and storage. That is, *programmers should be provided with uniform access to data whether it is private or shared, local or remote, temporary or persistent.* To effect complete uniformity in these respects, it is also necessary to mask the effects of failures due to conflicting concurrent accesses, distribution and storage management. The set of abstractions provided in the programming language to meet these requirements must be integrated with each other, yet individually applicable to classes of application data. These abstractions must be *selectively* intermixed as required to provide an exact match to the requirements of any particular application. Thus, for applications that involve temporary shared structures, the programmer should be able to specify support for sharing (and possibly distribution), without persistence. The ability to apply exactly the desired set of properties to individual application structures provides programming flexibility and run-time performance without sacrificing reliability.

How can these goals be achieved? There are three key questions to be addressed:

- What mechanisms should be developed or chosen to address the problems of reliable sharing, distribution and storage of data?
- How can the chosen mechanisms be smoothly integrated into a programming language interface, providing the desired level of transparency and selectivity?
- How can the desired combination of services be provided? That is, what kinds of operating system services, compiler and language features, and run-time support libraries must exist to implement the system?

1.3 Mechanisms for Reliable Controlled Sharing

Transactions (sometimes called *atomic actions* or *atomic transactions*) are the most commonly provided mechanism for reliable management of shared data. The transaction mechanism addresses both concurrency control and failure management for computations involving single or multiple objects.

Transactions are a structuring mechanism used by programmers to indicate a collection of operations that should (at least appear to) occur *atomically*. Why is such a mechanism necessary? The consistency of a set of objects could be compromised if either:

- two (correctly implemented) operations execute to completion, but their individual steps are interleaved in such a way that they each perform calculations using partially updated information of the other.
  or
- an operation fails before completion, leaving some objects updated and others not

Transactions provide three properties which together ensure consistency of data: (i) *serialisability*, (ii) *atomicity*, and (iii) *permanence of effect*. The first property ensures that the concurrent execution of transactions which access common data objects is free from interference, that is, a concurrent execution can be shown to be equivalent to some serial order of execution. The second property, *atomicity*, ensures that a computation can either be terminated normally (*committed*), producing the intended results or it can be *aborted* producing no results. Since the distribution of data introduces new sources of faults which can be difficult to cope with at the application level, e.g., communications failures, the atomicity property is especially important in distributed systems. The third property, *permanence of effect*, requires that final commit or abort decisions made for a transaction are irreversible and that any state changes produced by the end of the transaction are recorded on *stable storage*, a type of storage which can survive crashes with high probability.

While traditional transaction systems have been developed for use on a single machine, attempts to *distribute* transaction management capabilities over a network of computers are receiving renewed attention as networked computers become commonplace and customer demands drive applications designers to consider access to distributed data. Several transaction management systems have been successfully extended across machine boundaries although many research problems remain in improving performance, reliability, availability and administration of these systems.

1.4 Mechanisms for Managing Persistent Data

Traditionally, persistent application data has been stored in *files* or, for more complex data or data that was shared concurrently by many users, in *databases*. Because the transaction concept and development came from the database community, most transaction systems today are intimately linked to a particular database interface. It is from the operations of the database system that the transaction systems achieve their *permanence of effect* property.

The design of present–day database systems typically includes a collection of modules or processes which are organised by function e.g. *locking* for concurrency control, *logging* for recovery management, to provide the transaction facilities and management of persistent data. However, designs based on *files* or *relations* suffer from a weak data model and limited data manipulation functions. These weakness result in a view of data that is:

- lacking data abstraction
  Transaction processing systems modelled on relations or files rather than objects often expose details of data representation to all applications through the schema or file record layout. This makes applications unnecessarily vulnerable to "syntactic" changes in data representation and requires each application to reconstruct an appropriate interpretation for complex data items.

- passive
  The view of data as a passive "bag of bits" is especially evident in the traditional organization of transaction systems into transaction manager, concurrency–control manager, recovery manager, scheduler, etc. which are all independent of the actual data being manipulated. Each of these agents or processes maintains its own data structures for managing the data and is independently consulted in the access path to that data. When these agents all agree to allow a particular type of access to some datum, the data (bits) are

transferred to the application which may then operate on that data without regard to its higher level structure. Thus, while the database system *permits* some relationships to be expressed, it remains the responsibility of each application (programmer) to interpret the data correctly e.g., by performing "joins" that are meaningful. Application programs may perform invalid operations on the data items. Such invalid operations, if detected at all, will be detected only when an attempt is made to return the data to the DBMS for storage.

- Inflexible
  Policy decisions such as the locking protocol, recovery method and availability are centralized in the form of the agents mentioned above. These policies can rarely be varied on a type-specific or instance-specific basis. This lack of flexibility in control of policies results in operating inefficiencies and semantic problems. Since policies are decided once for all or at least once for each large group of items, the chosen policy must be very general to accommodate the range of items that fall within its scope. The application of such general policies precludes the use of type-specific performance optimisations and may conflict with semantics of some types, sacrificing representational accuracy for generality.

These weaknesses in conventional data models result from their inability to express the relationship between data objects and the operations which can be performed on them. There is an analogy here to early programming languages which failed to make the correspondence between the data structures used to implement some abstract type, and the operations which provided the behaviour of the object. While some *programmers* using these languages defined the data structures and operations together, the *programming languages* failed to provide a notation for expressing such relationships. Similarly, transaction operations for updating persistent data, usually provided in conjunction with some database system, are not usually well integrated into the programming languages from which they are used (cf. embedded SQL). This mismatch is due in part to the failure to accommodate the concept of persistent data in programming languages. Instead, programming languages typically provide procedural interfaces to operating system facilities like disk interfaces and file systems that provide primitive support for persistent data storage.

Since programming languages typically have no representation for persistent data, it is difficult to smoothly integrate a mechanism for manipulating persistent data into such languages. The result is a discontinuity in the semantics of data objects in which persistent data may be managed under the control of a transaction system, including concurrency control and failure management, but temporary data such as program variables are not. Individual programmers must make allowance for the differences in these interfaces. However, in the research community, there have been investigations into the development of programming models that include persistence as a *language* feature rather than an *operating system* feature. In such programming languages, the programming interface for data objects is the same, regardless of whether or not the data object "outlives" the execution of the program. That is, temporary and persistent data are treated uniformly. The adoption of persistence as a language feature leads to simpler programming models albeit with attendant complexity in compiler and run–time support.

There are several problems that arise as soon as one permits data to outlast a single program execution the most obvious of which is: How does one find the data again when the program (or some other program operating on the same data) is re–executed? Some kind of data naming scheme must be employed. In traditional database systems, the application program must identify an item of data by a *key* containing, for example, the database identifier (to locate the right database), the relation or set identifier (which identifies the right portion of the database and also conveys type information) and the "record key" or similar identifying information (to identify the individual item of data). All persistent data managers require some naming scheme, implying that some underlying mechanism must be provided to locate data efficiently. In a distributed system, this "location service" may naturally be distributed since the data will be distributed. The relationship of the location and naming services to the run–time support of persistent data is discussed in some detail in Chapter Three.

A second and more difficult problem arises from the difficulty of determining how to save program data in a way that preserves the semantics, especially the sharing semantics, of the data structures. For example, in a program containing a hash table which uses chaining to cope with overflow, saving a *persistent* hash table structure implies saving (a copy of) the elements of the hash chains. However, if the hash table entries consist of references to other persistent data objects, for example bank accounts, the saved hash table must *not* contain copies of the bank accounts, but references to them. These and other persistence support problems are addressed more fully in Chapters Three and Four.

The combination of ideas from the programming language community about persistent programming languages, from the database community about using transactions for concurrency control and failure management, and from the distributed systems community about managing distribution of data objects have the potential to provide a powerful toolkit for the development of reliable distributed applications. Applications programmers using such a toolkit should find it relatively easy to construct distributed applications that are reliable. To exploit the power of these tools, however, a uniform programming language interface must be developed.

1.5 Programming Language Options

To create a programming environment that smoothly integrates the chosen abstractions for persistence (persistence as a language feature), and concurrency control and failure management (transactions) requires careful attention to be paid to the stated goals of transparency and selectivity. The goal of transparency can be most easily met by inventing a new programming language which incorporates the desired abstractions into the programming language semantics, and developing language processors which invoke the supporting mechanisms as required to provide those semantics. Other implementations, for example, extending an existing language by pre-processing or compiler changes, could be nearly as effective if carefully developed. However, the overriding issue is of the abstractions that should be provided. This issue is especially evident in the goal of selectivity. What kinds of abstractions should be provided to minimize the burden on programmers?

Abstract Data Type (ADT) programming languages of the 1970's and 1980's, for example Ada or CLU, support a notion of encapsulation or modularity in which functions and variables which were part of the *implementation* of an ADT were hidden from the programmer using the ADT. The *specification* of a data type included only the functions of the interface, not their implementation. This idea of encapsulation has been carried forward in the next, evolutionary step in programming language design: *object–oriented programming languages*. Object–Oriented programming is an approach to program design and implementation that places primary importance on *objects*, that is the data in the design, and the behaviour of these data objects as specified by the operations they provide.

Object–oriented programming languages extend the encapsulation ideas of ADT languages with an explicit expression of the relationship between types. While encapsulation permits the programmer to express one kind of relationship between types, namely the "is–part–of" or aggregation relationship, object–oriented languages allow the expression of a second important relationship as well: the "is–a" or *inheritance* relationship. Objects are grouped into *classes* according to common behaviour (as defined by the operations provided). Each class definition specifies the programmer interface to the objects of that class and the relationship between classes – the inheritance hierarchy. Each object instance contains some variables (its instance variables) which are determined by the particular implementation of that object. The operations supported by an object have access to the instance variables and can thus modify the internal state.

In the object–oriented model of computation, encapsulation, inheritance and autonomy are fundamental principles. Encapsulation provides abstraction, and hides details of representation. Inheritance provides a mechanism for organizing classes of objects into a tree or, in the case of multiple inheritance, a directed acyclic graph, according to the generalisation / specialisation relationship. The autonomous nature of objects, whether actually realised by independent threads of control or not, is manifest in the concept that each object, or each class of objects, should "define" its own behaviour. That is, *all* of the behaviour of an object is defined by the operations which it exports. No direct external manipulation of the internal state is possible.

The ability to express in a "super–class" some common behaviour which is inherited by its "sub–classes" of objects enables the definition of classes that represent "properties" of objects. A new class of objects could inherit from multiple classes, acquiring the operations and properties of each of the parent classes. For example, classes such as *IsPrintable* or *IsSortable* convey attributes to objects which serve to classify them along alternate taxonomies. Thus, the class *Integer* might derive from the class *Number* to inherit basic mathematical operations, but also *Number* might derive from both *IsPrintable* and *IsSortable* in that numbers can be printed and sorted as well. This technique is widely applicable to other object-oriented languages and forms the basis of the reliability support described in this thesis.

Because object-oriented programming provides the best features of modular programming and augments them with explicit type relationships through the class hierarchy, this approach appears to offer a powerful aid to management of the complexities of reliable distributed applications development. The ability to combine, through inheritance, the behaviour of one class of objects with another class of objects proves to be valuable in providing the desired selective support for sharing, distribution and storage of data. Transaction classes can be developed to co-ordinate the operation of these different mechanisms with respect to managing failures and concurrent interactions.

1.6 Building a Reliable Distributed Object System

Adapting the object-oriented programming technique to the design of transaction systems requires a perspective shift in which data objects become "responsible" for themselves - rather than relying on external agents to control them. This autonomous, almost anthropomorphic view of data is characteristic of object-oriented designs. In the transaction processing field, this implies that functions like locking, transaction management, persistence and recovery are no longer provided by "manager processes" or "modules" but by the data objects themselves. Database systems for objects must be restructured to allow an abstract view of objects that are self-managing (although the *implementation* may still employ processes and modules for efficiency).

The individual properties of transaction management, concurrency control, persistence, recovery, and distribution must be separated to identify their individual requirements and interactions. Isolating these features allows applications designers to capture the exact level of functionality required, incurring cost only for the features that are actually used in each application level object. The ability to selectively apply these properties to different classes of objects is referred to as *selective transparency*. An important problem to be solved then is how to integrate the powerful, proven technology of modern transaction processing systems with the clear benefits of the object-oriented programming model to provide a programming interface that supports this selective transparency goal.

An object-oriented transaction system design will necessarily be structured in several levels. The overall system architecture, integrating the transaction system with the surrounding environment (operating system, communications network, storage systems) forms the highest level of the design. The structure of the run-time system for the classes and the client-server interactions with those class implementations forms another layer. Finally, a hierarchy of class definitions which can be individually applied to application-level objects provides the lowest level of the design - the programming interface to the system. These topics are addressed in turn in the chapters that follow:

- The applications programs execute in an environment, sometimes called a virtual machine, which provides services for locating, accessing, updating and controlling data and programs (and sometimes devices) that are outside the context of the application. Chapter Three proposes an architecture for this execution environment that addresses all of the *external* needs of reliable distributed applications. These are typically provided by a combination of operating systems support, hardware and software subsystems.
- The combination of services provided by the external support services and the programming language facilities provides the programming interface on which applications developers must build reliable distributed applications. Smooth integration of these facilities into the programming language (syntactically and especially conceptually) is essential. Chapter Four provides a description and rationale for the use of Object-Oriented programming techniques to capture these mechanisms and services in an integrated programming interface.
- Chapter Five outlines a series of example applications using the techniques described in Chapter Four and assuming the existence of a run-time environment as described in Chapter Three. The program fragments and discussion of these examples illustrate the usefulness of selective transparency in the development of reliable distributed applications.

- The final chapter concludes with a brief summary of the results and a look forward to implications for future operating systems, communications systems and programming languages.

1.7 Contributions of the Thesis

This thesis presents a novel architecture for distributed transaction processing in which the management of persistence, provision of transaction properties, and organisation of support services are all gathered into a unified design based on the object-oriented programming model.

Many mechanisms have been developed to address various aspects of reliability, persistence and distribution of objects. When applied in combination, these mechanisms may interact in fortuitous, or, more often, diabolical ways. This thesis explores the ways in which a set of known mechanisms for reliability, distribution, and persistence can be smoothly integrated, yet independently applied, by exploiting the power of object-oriented programming. The resulting programming interface should be powerful enough and simple enough to provide a significant improvement in the reliability of distributed applications constructed using this interface.

As this thesis will show, the integration of object-oriented programming concepts, persistent data as a language feature, and transaction processing mechanisms provides increased semantic information, better isolation of locking protocols and commit protocols, and independent management of persistence, recovery, concurrency, replication and migration. The result of these improvements is a better programming environment which simplifies the development of reliable distributed applications.

Thus, the thesis makes a contribution to the field of distributed transaction processing and databases, explaining how to apply the latest developments in programming languages to advantage. The thesis also makes a contribution to the field of object-oriented analysis, design, and programming in investigating the power and limitations of current object-oriented programming models.

2 Transactions on Persistent Objects

"An unreliable programming language generating unreliable programs constitutes a far greater risk to our environment and to our society than unsafe cars, toxic pesticides or accidents at nuclear power stations"
C. A. R. Hoare [Hoare 1981]

2.1 Introduction

The motivation for studying reliable, distributed, object-oriented programming with persistent objects is straightforward: programmers cannot cope with the enormous complexity of developing reliable distributed applications with present day tools. Although many individual mechanisms have been developed, a simple aggregation of those mechanisms would *increase* the complexity of the programming interface by confronting the programmer with a multiplicity of concepts, techniques and interfaces. What is needed is a programming model which encompasses some reasonable set of mechanisms in an integrated uniform interface. Furthermore, the facilities provided to the programmer must be cleanly separated and independently applicable to application-level objects.

As will be seen in later chapters, this thesis proposes just such a flexible, uniform interface – a declarative, object-oriented programming interface for the development of reliable distributed applications. However, to establish the background leading to this result, it is necessary to consider the requirements of a programming environment for reliable distributed programming, and the mechanisms that might support those requirements. The previous chapter identified a possible set of mechanisms:

- A transaction mechanism as the basis for achieving reliability; support for nested transactions removes constraints on the use of transactions in modular software development

- A remote procedure call mechanism, augmented with naming and locating services, for supporting distribution;
- A persistent programming mechanism instead of other less transparent mechanisms, such as files or relational database systems, for uniform treatment of persistence.

This chapter explains the rationale for this choice of mechanisms, including the choice of object-oriented programming as an integration vehicle, and compares the facilities provided by the proposed mechanisms with other systems which have addressed some aspects of the problems of reliable distributed programming.

In the first half of this chapter, each of the issues of reliability, distribution and persistence are addressed in turn. A framework of requirements is established and a range of mechanisms to support those requirements is discussed. Particular consideration is given to the interaction of object-oriented programming with the various mechanisms. The second half of the chapter surveys related systems, not to explore their many merits, but to highlight the different tradeoffs that have been made in those systems and to evaluate their capabilities and deficiencies in light of the framework of requirements established here.

2.2 Achieving Fault Tolerance with Transactions

2.2.1 Fault Tolerance

Although the term "reliable" colloquially implies trustworthiness, predictability, security and safety, within the research community studying fault tolerance, *reliability* has a more precise technical meaning: "the ability of the system to deliver its normal service" [Lee and Anderson 90] [Laprie 89]. Fault avoidance, for example, through improved specification and design methods and tools, provides one means for improving the reliability of a system. However, to the extent that faults can never be completely avoided, fault tolerance remains necessary as a means of improving the reliability of a system.

Following the terminology of Melliar-Smith and Randell [Melliar-Smith and Randell 77], the following terminology is used throughout this thesis:

- A *fault* is the mechanical, human or algorithmic cause of an error.

- An *error* is an item of information designating the part of the state which is erroneous or "incorrect".
- An *erroneous state* is an internal state of a system such that further normal processing may lead to a failure.
- A *failure* is an event that occurs when a system does not perform according to its specification.

The precise use of this terminology will be especially important in the discussion of recovery mechanisms. If the meaning of reliability is to deliver normal service in spite of faults, it is important to identify explicitly the nature of the faults that can be tolerated. Attempts to classify faults [Powell 91] [Shrivastava *et al* 90] [Bernstein *et al* 87] [Ezhilchelvan and Shrivastava 86] have helped to identify the range of faults that can occur in distributed computing systems. These classifications differ in that some attempt to classify the faults according to their sources; others classify according to the consequences of the faults, i.e., the severity of failure that would result if the fault were not properly treated. These taxonomies of possible faults provide a systematic framework against which a range of fault tolerance techniques can be applied and evaluated.

Program faults, in the design or implementation of algorithms, may occur in any type of programming although they are likely to occur more frequently in complex programming environments such as those presented by distributed systems. The incidence of design faults may be reduced by software engineering methods such as formal specification / verification. The effects of design faults may be minimised by the application of design fault tolerance techniques such as N-version programming [Avizienis 85] [Knight and Leveson 86] or recovery blocks [Horning *et al* 74] [Anderson and Kerr 76] [Lee *et al* 80]. Beyond design faults, there exist many other potential sources of faults in a distributed system such as operator faults, programming faults and physical hardware faults. Permanent faults, such as those exhibited by hardware components suffering physical deterioration, can be addressed by spatial redundancy – e.g., disk media faults can be masked by appropriate use of disk mirroring. Transient faults are typically treated at a higher level – e.g. temporary communications loss can be masked by reliable communications protocols using temporal redundancy. Program faults are always permanent with respect to a single version of a program, but may cause transient, intermittent or permanent failures of system components. For example, a program fault in the operating system may cause intermittent system crashes; a program fault in a software protocol engine may cause occasional communications failures due to a failure to meet the protocol specification.

2.2.2 Transactions

To tolerate component faults, whether due to hardware or software faults or human mistakes, some kind of error recovery mechanism must be employed. For information processing applications, as opposed to, for example, process control or other real-time activities, transactions [Spector and Schwartz 83] (also known as atomic actions or atomic transactions) are a widely accepted software structuring tool for demarcating the scope of error recovery for high-level fault-tolerance:

"... *regardless of advances in hardware, we believe atomic actions are necessary and are a natural model for a large class of applications. If the language / system does not provide actions, the user will be compelled to implement them, perhaps unwittingly reimplementing them with each new application, and may implement them incorrectly*" [Liskov and Scheifler 83].

Some radically different, non-transaction-based approaches have been proposed to address the problem of maintaining consistency despite faults (cf. ISIS below). While these approaches have been successfully employed for certain application areas, transactions are the most widely used, best understood approach in database systems supporting information processing.

Transactions provide the properties of (i) *concurrency control*, (ii) *permanence of effect, and* (iii) *atomicity*:

- **Concurrency control** means that concurrent operation invocations from different transactions accessing common data objects are free from interference. One way to ensure that interleaved transaction steps are free from interference is to ensure that the schedule of interleaving is serialisable [Bernstein and Goodman 81]. Serialisability means that a concurrent execution of transaction steps can be shown to be equivalent to some serial order of execution of those steps. Some form of concurrency control policy, such as that enforced by two-phase locking [Eswaren *et al* 76], is required to ensure the serialisability property of transactions.
- **Permanence of effect** means that any changes produced in the states of persistent objects during the execution of a transaction are recorded on *stable*

storage, a type of storage which can, with high probability, mask faults such as processor failure or power failure.

- **Atomicity** means that a computation executing within the bounds of a transaction will either be terminated normally (*committed*), producing the intended results (and intended state change to the data objects involved) or it will be *aborted* producing no results and no state change to the data objects; no intermediate states will be visible outside a single transaction even if failures occur. Whenever a fault occurs that cannot be masked, atomic recovery of the states of objects involved in a transaction is invoked. Typical failures causing a transaction to be aborted include process or machine "crashes" and communication failures such as the continued loss of messages. A *commit protocol* is required during the termination of an atomic action to ensure that either all the objects updated within the action have their new states recorded on stable storage (committed), or, if the atomic action aborts, no updates get recorded [Bernstein *et al* 87].

Transaction *begin* and *end* operations define the boundaries between which these properties take effect. That is, the sequence of operations that is performed between the start and end of a transaction is atomic with respect to all other accesses to the objects involved. Similarly, the transaction *end* determines the point at which intermediate state changes to the objects involved in a transaction become permanent. Each of these properties is examined in more detail in the sections that follow.

Concurrency Control in Transactions

The purpose of concurrency control in the behaviour of transactions is to ensure object state consistency in the event of concurrent access. Although this property is generally enforced by a serialisability constraint on transactions, serialisability itself is not a goal of transaction processing systems; the goal is to achieve maximum concurrency without compromising correctness. However, the correctness of an arbitrary interleaving of transaction steps is difficult to determine directly. Some non–serialisable interleavings of transaction steps are correct in that the resulting object states are entirely consistent. Unfortunately, no general method has been devised for distinguishing these correct, non–serialisable interleavings from inconsistent ones. Although serialisability sometimes restricts concurrency to something less than the maximum achievable, the serialisability property is a simple way to isolate the interactions of transactions and simplifies reasoning about their correctness. The effect of enforcing a serialisable ordering on transaction steps and hence excluding some correct, but non-serialisable interleavings is to reduce potential concurrency resulting in reduced performance. In an attempt to increase concurrency, some researchers have investigated relaxing the serialisability constraint to include a larger class of schedules for transaction steps, for example, by employing multi-level serialisability constraints [Weikum 91]. System R [Gray *et al* 81] was one of the few systems ever actually developed which employed a notion of multi-level serialisability to increase concurrency.

Permanence of Effect in Transactions

The requirement for permanence of effect for transactions implies that the results of transactions can be recorded in some way that survives subsequent system (or media) failures. The concept of *stable storage* provides just such high reliability permanent storage. Implementation techniques for stable storage using multiple disks or stable RAM have been proposed in [Lampson and Sturgis 79] and [Banâtre *et al* 83] among others. Stable storage may be employed in the implementation of the transaction system, to provide permanence for transaction state and operation logging, as well as at the application level, providing permanent storage for application program objects.

Atomicity in Transactions

To provide atomicity, a transaction system must provide some co-ordinated form of recovery for objects that were modified in the course of execution of a transaction in case that transaction is aborted. The two principal techniques that can be used for error recovery are *backward error recovery* and *forward error recovery*. The type of recovery that is employed to implement atomicity for the transaction abstraction will depend on the nature of the objects and the extent to which damage resulting from faults can be predicted.

In forward error recovery, the recovering objects perform some specific corrections to their state in an attempt to remove errors and hence permit the system to continue operation without a failure occurring. The changes do not usually or necessarily return the individual objects or the overall system to some prior state. The use of forward recovery is dependent on the ability of the programmer to anticipate the kinds of faults that could occur, the nature and extent of the resulting errors, and appropriate responses to ensure a consistent system state.

Backward error recovery is a more general technique which can accommodate the effects of unanticipated faults. In backward error recovery, the objects involved in a transaction are restored to some consistent prior state. This restoration of a prior state may be effected either by replacing the entire state of the object with a previously captured snapshot of the prior state (state–based recovery) or by logging a record of all operations performed since the start of the transaction and "undoing" the effects of those operations by applying inverse operations (operation–based recovery).

Backward error recovery is inappropriate when the effects of a transaction cannot be reversed by restoring the state of the objects involved in the transaction. For example, if a rocket was fired as a result of some action taken in a transaction and the transaction was subsequently aborted, recovery of the state of the rocket will be impossible. Gray has called such irreversible operations "real" because they involve objects in the real world [Gray 80] although there also exist abstract operations from which is it difficult to recover. Recovery from "real" operations may require forward error recovery or manual intervention.

Recovery techniques, especially state–based recovery techniques, are related to persistence because of the recovery requirement to store a "copy" of the state that will survive a failure. In the case of persistent objects, a copy of the persistent state may already exist on some permanent media providing a ready snapshot of the prior state, hence the relationship between the otherwise separate concepts of persistence and recovery.

2.2.3 Transactions as a Tool for Fault Tolerance

The predominant use of transactions for fault–tolerance in information processing applications is perhaps attributable to the simplicity of the interface and the power of the mechanisms, encompassing at a stroke concurrency–control and atomicity. Extending a transaction processing environment across machine boundaries offers the potential to expand the power available to applications beyond the capabilities of any single computer system. For the purposes of this thesis, it is sufficient to consider a distributed system to be any collection of two or more computers (nodes), connected by a communication subsystem (for example, a local area network). To support transaction processing in such a system, at least one node must posses some form of stable storage. Faults in the communication subsystem may result in problems such as lost, duplicated or corrupted messages. However, transient communication faults such as these can be dealt with by well known network protocol level techniques, so their treatment will not be discussed further. Permanent communications faults in a distributed system such as network hardware failure, network partitioning or continued loss of messages can be treated at the transaction level using recovery techniques as described above.

2.3 Distribution

Distribution can offer more than just increased computing power. To improve availability and reliability, data and computations can be replicated on multiple machines in a distributed system. This replication can be used to permit continued operation in the event of machine failures and hence availability can be increased. Going a step further, if the results of independent replicated computations are compared, certain kinds of software and hardware errors can be detected and masked, resulting in improved reliability. Distribution may also facilitate sharing of scarce network–wide resources, for instance, to balance the computational load across computing resources. However, the benefits of distributed systems are not achieved without cost.

The primary cost incurred in exploiting the potential benefits of distributed systems arises from the increased complexity of the programming interface for such systems. Unless care is taken in the design of the application programmers' tools, programming a distributed system can be intolerably difficult. Programmers must not be burdened with of the complexities of a distributed environment in addition to the difficult tasks they already face in the development of complex applications. To relieve programmers of the additional burden of distribution requires tools or system facilities which make distribution *transparent.*

2.3.1 Distribution Transparency

The concept of "transparency" has been proposed to describe distributed systems which can be made to behave like their non–distributed counterparts

[ANSA 89]. This transparency property minimises the difference to the application developer between programming for a distributed systems environment and programming for a single machine. There are several complementary aspects of transparency:

- *access transparency* mechanisms provide a uniform means of invoking operations of both local and remote objects, concealing any requisite inter–node communications;
- *location transparency* mechanisms conceal the need to know the physical location an object, making it sufficient to be able to name an object to access it;
- *migration transparency* mechanisms support the movement of objects from node to node, for example, to improve performance or fault–tolerance;
- *concurrency transparency* mechanisms ensure interference–free access to objects in the presence of concurrent invocations;
- *replication transparency* mechanisms increase the availability or reliability of objects by replicating them while concealing the intricacies of maintaining consistency among the replicas;
- *fault transparency* mechanisms help exploit the redundancy in the system to mask faults where possible and to effect recovery measures when faults cannot be masked.

Access transparency and location transparency are closely linked, but are not the same. Access transparency implies that local and remote operation invocations are specified identically in application programs. That is, access transparency implies that:

- no extra information need be specified to access remote objects
- it is not apparent from the invocation syntax whether a local or remote object is being operated upon
- the semantics of an operation are unaffected by whether the operation is local or remote

In contrast, location transparency requires that the programmer need not be aware of whereabouts of an object used in the program. This implies that the location of the object can somehow be derived from the name of the object, for example, through a *name server* or *location server.* In a system that lacked location transparency, the programmer might be required to supply a host node name when first referencing a remote object, although subsequent operation invocations on the object might be *access transparent* in that they had the same syntax and semantics as local operations.

As an example illustrating the difference between access and location transparency, consider a network file system in which a remote file can be operated upon like a local file, but the file name includes the host name on which the file currently resides e.g., host::foo.dat. Such a file system interface is access transparent because operations like read and write are identical for all files. However, the interface is not location transparent because it is the responsibility of the user or programmer to identify the location of every file to be opened. Compare this with a distributed file system that maintains an up-to-date distributed database which maps every file name to the present location of that file. The user or programmer of this second system need only specify the name "foo.dat" to access the file anywhere in the network. This second system supports location transparency as well as access transparency.

Access transparency is linked to the operation invocation mechanisms of programming languages. Since the goal of access transparency is to make the specification of operation invocations uniform regardless of whether or not the invoked object is local, the syntax and semantics of remote and local invocation must be made identical. Since many object-oriented languages define operation invocation to be synchronous, *remote procedure call* (RPC) [Nelson 81] is a natural communications paradigm to adopt for the support of access transparency in object-oriented languages. Access transparency is normally provided by integrating an RPC pre-processor into the program development cycle [Birrell and Nelson 84].

An RPC pre-processor parses the interfaces to application objects and produces "stub" code for both the client, the application program, and the server, object implementation. The stub code on the client side converts operation invocations into messages and forwards them to the server. The stub code on the server side unpacks the messages and converts them into operation invocations on the actual object implementations. Results of the invocation are packed into messages by the server stub code and transmitted back to the client. Finally, the client stub code unpacks the reply messages and returns the results to the application. Note that some kinds of operation invocation semantics are difficult to provide across machine boundaries. For example, some languages require parameters to be passed by name or by reference. These semantics are difficult to achieve across address space boundaries making some programming languages more difficult to distribute than others. Object-oriented programming languages are generally less troublesome in this regard because they enforce encapsulation of object data and hence support controlled, functional access to data rather than direct, address-space-dependent reference semantics.

Location transparency can be provided by a location or naming service such as the ANSA Trader [ANSA 89] or Sun NIS [Sun 88]. A variety of naming, binding and caching strategies are possible to manage this service efficiently.

The property of migration transparency requires both access and location transparency. If a program does not specify the location of the objects upon which it is operating, and the syntax and semantics of local and remote operations are uniform, then objects could move transparently from one node to another. In the simplest case, objects may move while not involved in any transactions. It is also possible to move objects between individual operations of transactions or even (in limited cases) during operation invocations on those objects, but these moves require additional context information to be transmitted with the object state.

Objects might be migrated for a number of reasons. One obvious use of migration would be for load balancing, but efficient load balancing by object migration has been shown to be problematical [Eager *et al* 88]. However, there are other reasons, such as reliability, security or logistics, why one might like to move objects during a system's operation [Liskov and Scheifler 83]. Migration transparency simply states that this migration can occur without explicit programmer intervention.

Concurrency transparency, one of the principal properties of transactions, frees the programmer from concern over the effects of interacting operations. Providing concurrency control for shared objects is a serious complication for application programmers even in centralised systems, hence transparent concurrency control is desirable. Since, in distributed systems, concurrency–control state is necessarily distributed, the problems of serialisability and deadlock management are severely complicated. The lack of global state information for the distributed system makes it difficult to determine the overall state of a set of objects e.g., to detect deadlock. The lack of global control makes it difficult to correct errors when they are detected e.g., to break a deadlock. Finally, inconsistencies may arise because individual components of a distributed system, such as a single node, can fail while the remaining components continue processing a transaction. For example, a node may fail while some application running on that node is holding locks on remote objects. When the failed node recovers from the crash, there may be no record of which locks were held before the crash. These difficulties compound the problem of developing reliable distributed applications and hence, in distributed systems, concurrency transparency is a practical necessity.

Replication transparency frees the programmer from the necessity of knowing which of the objects used in a transaction are replicated. Replicated objects must be managed through appropriate *replica–consistency protocols* to ensure that the object copies remain mutually consistent. Such protocols can be integrated within transaction systems as discussed in [Bernstein *et al* 87] to provide replication transparency. In transparent replication schemes, programmers specify the desired level of availability or reliability for an object, leaving the creation and initial placement of replicas and the consistency and population management of the replica group to the distributed system. Note that, like migration transparency, replication transparency implies access and location transparency.

Fault transparency mechanisms are particularly important in distributed systems where the possibilities of node crashes and communications faults significantly increase the range of failures which could affect an application. In a fault transparent system, a programmer need only indicate the bounds of a recovery region. When faults occur, the underlying system transparently recovers the state of the objects which have been modified within the bounds of the recovery region. The abstraction of a transaction is, among other things, a means of defining the bounds of a recovery region.

Each of the six types of distribution transparency described above can contribute to the development of reliable distributed applications. However, not all such applications will need all types of transparency and some important aspects of the programming problem are not addressed by any of the above. In particular, the provision of a persistent programming mechanism, while somewhat independent of distribution aspects, is another important facet of the programming interface for reliable applications development. Taken together, the support for distribution delineated in the section above and the support for persistence described in the next section form a framework of requirements against which a distributed system could be developed.

2.4 Persistent Objects

Traditionally, persistent or permanent data has been stored in *files* or, for more complex data or data that was shared concurrently by many users, in *databases*. Basic research into persistent programming languages such as PS–Algol [Atkinson *et al* 81] and Napier [Morrison *et al* 89] have extended the traditional concepts of program state to include persistence as an orthogonal property. The following taxonomy of the uses of persistence and the mechanisms that support them is given in [Atkinson *et al* 90]:

Persistence is a property of data which describes "...the period of time for which the data exists and is usable. A spectrum of persistence exists and is categorized by

1. transient results in expression evaluation
2. local variables in procedure activations
3. own variables, global variables and heap items whose extent is different from their scope
4. data that exists between executions of an application program
5. data that exists between various versions of an application program
6. data that outlives an application program.

The first three persistence categories are usually supported by programming languages and the second three categories by the DBMS, whereas filing systems are predominantly used for categories 4 and 5."

Understanding the spectrum of persistence and observing the dichotomy that currently exists in the treatment of objects with differing lifetimes helps explain some of the difficulties programmers face in managing persistent objects using conventional programming languages and tools. As Atkinson *et al* have observed, the conventional split between temporary and permanent data causes difficulties for programmers, employing programming language operations and semantics for the former and file or database system operations and semantics for the latter. Other research efforts have included the more involved concept of a *persistent object* in the sense of object-oriented programming languages such as Beta [Agesen *et al* 89], Procol [van den Bos and Laffra 89], Galileo [Albano *et al* 88], Trellis/Owl [O'Brien *et al* 86], and E [Richardson and Carey 88]. In some cases, persistence has been provided in existing object-oriented languages by creating libraries of classes with operations that specifically support persistence, for example the NIH class library [Gorlen *et al* 89] and Arjuna [Dixon *et al* 87] [Dixon 88]. The distinction between operations on traditional state variables and *objects* is more fully discussed in section 2.6.

The results of the research into persistence as a programming language feature lead persuasively to the conclusion that both temporary and persistent objects can, and should, be treated uniformly. The simplification of programming in a language that treats the full spectrum of persistence in a uniform way should reduce conceptual overhead for programmers and result in more reliable programs. The term, *persistence transparency,* captures the concept of uniform treatment of persistent and transient objects analogous to distribution transparency for local vs. remote objects. Although some notion of persistence is implicit in the "permanence of effect" property of transactions, persistence is a separate concept.

2.5 Transparencies and Properties

Each of the transparencies described above is identifying a dimension along which objects may be categorised. For example, concurrency transparency refers to uniform treatment of objects whether or not they are shared, while persistence transparency refers to uniform treatment of objects regardless of how long they last. Since different classes of application objects have differing requirements, there is a need to allow programmers to apply these properties selectively.

For an example of the importance of selective application of properties to objects, consider the actions of a free storage manager like the unix facility, malloc. The objects composing the data structures of the storage allocator will need to be recoverable since any storage allocated during the execution of a transaction should be returned to the free storage pool if that transaction aborts. Access to the objects which implement the storage allocator may need to be concurrency controlled if multiple transactions might simultaneously access the same heap. However, the objects which maintain the structure of the heap may not need to be persistent. Thus, the objects used in the implementation of the storage allocator might require recovery and concurrency control properties, but not persistence.

Another class of objects that has similar requirements is network routing tables. Network routing tables might need to be concurrency controlled, because they are potentially concurrently accessed by many different users. They might need to be recoverable, to maintain consistency when transactions that update them abort, but they might not need to be persistent since they are normally acquired by interaction with neighbouring sites during system startup. Whatever combination of properties is applied to an object, the transparencies ensure that the object can and will be treated uniformly. How do these properties and their corresponding transparencies relate to transactions?

Traditional notions of transactions incorporate persistence, concurrency control and recovery in the definition of "transaction". The relationship of the transaction to the other objects involved in the transaction is one of boundary definition. A transaction defines the duration for which locks are held (in strict locking schemes), the point at which updates to persistent objects are recorded, and the recovery region over which a recovery point is active (for backward recovery schemes). While this use of transactions for demarcating the boundaries of transparency behaviours is beneficial, traditional transaction systems have failed to separate these transparencies from each other or from other kinds of transparency which are more specifically related to distribution such as access, location, replication and migration transparencies. Selective application of transparencies generalizes the transaction concept by separating it from specific transparencies and by employing the transaction as a general boundary manager for all transparencies.

If transactions are serving to delineate boundaries over which the distribution and persistence transparencies operate, it is reasonable to ask what happens when no transaction is present. Our model is that application programmers create and start transactions, perform a sequence of operations on a set of objects, then commit or abort the transactions. In such cases, the transaction termination serves as a boundary to the persistence and distribution mechanisms, indicating that some action must be taken which is appropriate to the termination of the transaction. For example, locks can be acquired on an object in the course of execution of a transaction. These locks will be released "automatically" when the transaction commits or aborts. But what happens when the same object is operated upon by an application that is not executing a transaction? Locks should still be acquired to ensure consistent access to the state of the object, but will not be automatically released until the object reaches the limit of its extent (in the programming sense) i.e., it ceases to exist in the application program. The ability to employ either "transaction–oriented" or non–transaction–oriented clients with the same objects is an additional form of transparency which may be termed, *transaction transparency*.

For the distribution and persistence transparencies to make sense in the absence of transactions, some other boundary determination must be used. Two obvious candidates are: 1) explicit boundary determination and 2) behavioural boundaries coincident with the lifetime of the object in the application. For example, an application can define its own recovery regions by explicit operations to start and end a recovery region. Similarly, an application could explicitly end a persistence region by force–writing the current state of the object to stable storage. A more subtle and perhaps more useful means of determining the boundaries over which the transparencies apply would be to use the variable extent rules of the programming language to mark the start and end of the regions., For example, when a persistent object first comes into scope (extent) in the application program, its state can be automatically loaded from stable store. When the object extent has ended, the persistent state can be updated on stable storage. No explicit action would be required on the part of the application programmer to manipulate the persistent state of the object. Hence, *transaction transparency* allows an additional degree of flexibility in the selective application of properties to classes of objects.

Even the most basic properties implied by access and location transparency may be selectively applied. In a distributed system supporting transparency, remote access and location transparency are basic properties which all shared or persistent objects must possess. However, not all objects in a distributed system are shared or persistent. Local, transient objects such as temporary variables in an application or partially computed results need not have their locations recorded anywhere because their location can always be known to all of their clients without any run-time lookup; such objects need not be available for remote access because they are always accessed locally. The concept of transparency still applies: objects which are local are invoked in the same way with the same syntax and semantics as objects that are (potentially) remote. However, the combination of properties assigned to individual classes of objects will vary.

The requirements for an effective development environment for reliable distributed applications then must include the ability to apply transparencies selectively to objects. It must be possible to apply any combination of the properties implied by the various forms of transparency, while maintaining uniform syntax and semantics of operation invocation regardless of the particular combination selected. The principal means by which programmers attribute properties to objects is through programming language constructs and facilities. As discussed in Chapter One, object-oriented programming may provide a particularly appropriate mechanism for selective attribution of some of these properties to objects through the use of multiple inheritance. The next section provides a discussion of the fundamentals of object-oriented programming and, in particular, the concept and application of multiple inheritance.

2.6 Object-Oriented Programming

Object-orientation is an approach to program design and implementation that places primary importance on *objects*. Objects are grouped by an application designer into *classes* according to common behaviour (as defined by the operations provided). Each class definition specifies the programming interface to the objects of that class and one important aspect of its relationship to other classes – the inheritance hierarchy. Each object instance contains some variables (its instance variables) which are determined by the particular implementation of that class of objects. The operations supported by an object have access to the instance variables and can thus modify the internal state of the object. It is assumed that, in the absence of failures and concurrent invocation of operations on an object, the invocation of any of the operations of the object produces consistent state changes to the object.

Object-oriented programming languages allow programmers to express an important relationship between object classes, the "is-a" or inheritance relationship. For example, in an object-oriented language one can define a class, creatures-that-fly, that represents an abstraction of a number of subclasses like flying-insect or bird. Each of these classes can be further refined to more concrete subclasses such as bumble-bee or macaw (see Figure 2-1).

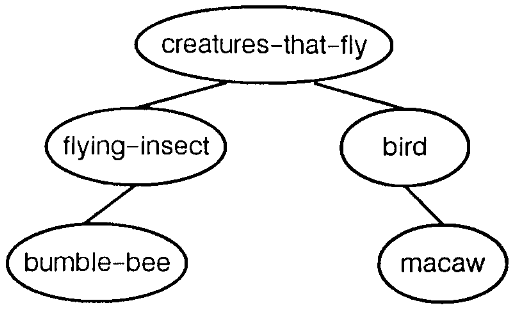


Figure 2-1: A simple class hierarchy

Associated with each class, such as creatures-that-fly, is some set of properties, *operations*, that define some aspect of behaviour which is common to all the members of that class and, in some form, to all subclasses. For example, all creatures-that-fly have some positive, even number of wings. The operation, number-of-wings, could be defined for the abstract class, creatures-that-fly, thus abstracting a common feature of all flying creatures.

A similar class hierarchy could be defined for all creatures-that-swim (see Figure 2-2). The abstraction, creatures-that-swim, might capture features common to all such creatures, such as the type(s) of water in which they swim: fresh water, salt water or both.

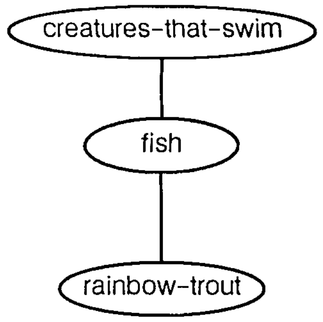


Figure 2–2: An example of single inheritance

These examples illustrate the use of single inheritance – each class is derived from at most one other class. A derived class is termed a subclass and the the class from which it is derived is termed its superclass. What happens now when one tries to incorporate into these two single inheritance hierarchies a class of creatures called, ducks? Ducks can swim *and* fly. In fact, ducks are a subclass of a more general class of creatures called, waterfowl, that can both swim and fly. One could define a new "top level" class called, creatures-that-fly-and-swim. However, such a solution is undesirable for two reasons:

1. The number of class definitions explodes combinatorially with the number of properties to be modelled.
2. The potential for sharing of specifications and possibly implementations that might have occurred between the classes, creatures-that-fly class and creatures-that-swim-and-fly is lost.

To adequately model the abstraction of waterfowl within the hierarchies already developed and preserving sharing, requires *multiple inheritance*, that is, the ability to derive a new class from more than one other class, hence incorporating the behaviour of all of the superclasses in the subclass (see Figure 2–3).

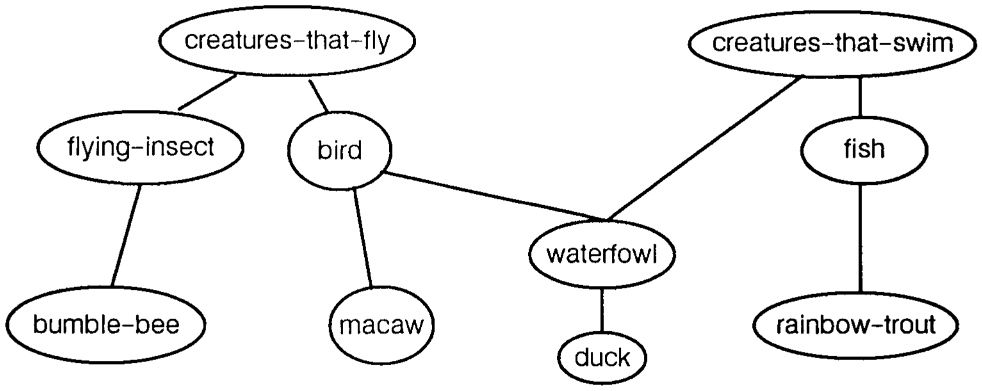


Figure 2–3: An example of multiple inheritance

While Figure 2–3 may represent a naive taxonomy of the species, it serves to illustrate the power of multiple inheritance to model the relationships between abstract classes of objects. Furthermore, the top two classes in Figure 2–3, creatures–that–fly and creatures–that–swim, represent abstractions, not of whole disjoint subclasses of creature, but of particular aspects of creatures in general. One could imagine defining creatures–that–walk, creatures–that–burrow, etc. With a range of such classes, it would be possible to classify, for example, muskrats (swimming, walking, burrowing), elephants (walking) and hummingbirds (flying, walking). Such a categorisation by properties is not the only way to organise species data, but it is useful because it gathers subclasses of objects together according to common behaviours.

The ability to express in a superclass some common behaviour of several subclasses of objects is equivalent to the definition of classes that represent "properties" of objects. This idea was explored in depth in the development and application of "mixins" in the Flavors System [Weinreb and Moon 81] [Moon 86] and "traits" in the Star system [Curry *et al* 82]. In these systems, classes of objects were defined that simply added some characteristic to a class derived from them. A new class of objects could inherit from multiple classes, acquiring the operations and properties of each of the parent classes. This technique is applicable to other object–oriented languages and provides a mechanism for selective attribution of properties to objects. Thus, as Chapter Four will explain in detail, multiple inheritance can be used to represent properties of object classes supporting reliable distributed programming.

2.7 Objects and Transactions in Distributed Systems

An object–oriented programming language supporting multiple inheritance to "mix and match" object properties could provide a suitable basis for reliable distributed programming by enabling an application programmer to selectively apply properties to objects. Given a set of classes that provide the properties of persistence, concurrency control, and recovery, multiple inheritance could be used to derive application classes that exhibit persistence, concurrency and fault transparencies. Building an appropriate support environment including communications, location services and stable storage could provide the remaining distribution transparencies. These support facilities, their organisation and relationship to transaction management are discussed more fully in Chapter Three.

Distribution transparency and persistence transparency have been treated in various ways in previous works, but rarely with a clear separation of properties. As with traditional transaction systems, previous research efforts have generally failed to provide the ability to selective apply properties to objects, while retaining the overall transparency of operation invocation syntax and semantics. Other researchers in this area have made differing assumptions about requirements and have focussed on different aspects of the general problem of programming distributed systems. In the next section, related systems are examined with particular emphasis on their relationship to the research reported in this thesis. In particular, for systems supporting reliability in some way, attention is paid to the ability of programmers to separate the concerns of recovery, persistence and concurrency control from each other and from other aspects of distribution transparency as defined above.

2.8 Related Systems

There are many aspects to the development of a system for reliable distributed programming. Not surprisingly, there are many related research efforts that bear on different aspects of the overall problem. Each of these efforts has contributed knowledge, insight, and experience in addressing the issues arising from reliability, distribution and persistence. However, none of the systems described below has tackled the difficult, but valuable issue of selective transparency.

One approach to the development of a programming environment for reliable distributed applications is to construct an object-oriented interface to a conventional transaction processing system. In some such systems, object-orientation has permeated several layers of the system, but the rigidity of the underlying system still shows through this object-oriented veneer. As a consequence, these systems are unable to grow and evolve naturally in an object-oriented framework because the fundamental properties of transactions are represented by collections of subroutines rather than by objects. Other approaches incorporate the object-oriented paradigm at lower levels, providing a smooth progression of functionality for application level programs. A brief study of the most closely related systems is given in the next five sections, 2.8.1 to 2.8.5.

Distributed programming languages / environments such as **Emerald** [Black *et al* 87] [Raj *et al* 91], **Distributed Smalltalk** [Bennett 90] **Amber** [Chase *et al* 89], **Argus** [Liskov 84] [Liskov 88], and **Comandos** [Marques and Guedes 89] have made considerable progress in the provision of a truly object-oriented distributed programming system addressing issues of distribution transparency and migration. However, in supporting persistence and reliability, these systems do not generally separate issues of recovery, concurrency control and persistence, preferring to provide "atomic" data type declarations which are all-or-nothing mechanisms.

Persistent object programming languages such as **Napier, Galileo, Procol** and **E** have cleanly incorporated persistence with the object model, but generally consider distribution, migration, reliability and availability as an afterthought, if at all. Section 2.8.6 summarises related research projects into programming languages for distributed systems.

Some attempts have been made to address the difficult problem of maintaining a consistent view of data across heterogeneous machines in a network. For example, **Ovid** [Hollberg *et al* 90] is an "open distributed support environment" supporting distribution in a heterogeneous network. More generally, distributed operating systems have been extended in various ways to provide primitive support for objects and / or to provide extended support for reliability, usually through some kind of transaction mechanism. Research efforts in the area of distributed operating systems are described in section 2.8.7 below.

Object-oriented data base systems have addressed many of the problems of concurrency control, persistence and recovery of objects, but are generally weak in the area of distribution especially with respect to migration, replication and location transparency. An excellent summary of current research in this area is given in [Zdonik and Maier 90]. The subject of object-oriented database systems is not treated further in this thesis.

In each of the systems examined below, there has been a somewhat different focus. The following sections briefly outline these related projects, describing their relevance to the present work and their shortcomings in addressing the goal of reliable distributed processing within the framework described in this chapter.

2.8.1 Arjuna

Arjuna [Dixon *et al* 89] [Parrington and Shrivastava 88] [Parrington 90] [Shrivastava *et al* 91] is an object-oriented programming system that provides a set of tools for the construction of fault-tolerant distributed applications. Arjuna provides nested transactions for structuring application programs. Transactions operate on objects, which are instances of classes, by making use of remote procedure calls (RPCs).

Three properties were considered highly important in the design and implementation of Arjuna:

- Integration of mechanisms: a fault-tolerant distributed system requires a variety of system functions for naming, locating and invoking operations upon local and remote objects and also for concurrency control, error detection and recovery from failures. These mechanisms must be provided in an integrated manner such that their use is easy and natural.
- Flexibility: these mechanisms should also be flexible, permitting application specific enhancements, such as type-specific concurrency and recovery control, to be produced easily from existing default ones.
- Portability: the system should be easy to install and run on a variety of hardware configurations.

The computational model of transactions controlling operations upon objects provides a natural framework for incorporating integrated mechanisms for fault-tolerance and distribution as has been discussed in section 2.7. In Arjuna, these mechanisms have been provided through a number of C+ + classes. Arjuna is novel with respect to earlier fault-tolerant distributed systems in taking the approach that every major entity in the Arjuna system is an object. Thus, Arjuna not only supports an object-oriented model of computation, but its internal structure is also object-oriented. This approach permits the use of the inheritance mechanisms of object-oriented systems for incorporating the properties of fault-tolerance and distribution in a flexible way, permitting type-specific implementation of concurrency control and recovery for objects. Arjuna has been implemented without any changes to the underlying operating system or language, employing single inheritance to organise the "property" classes for management of persistence, recovery and concurrency control. Persistence and recovery are treated together in one class and concurrency control in another which is derived from the first.

The Arjuna system was the starting point for the work reported in this thesis and therefore the two systems bear a strong resemblance to each other. However, Arjuna fails to provide adequate support for selective application of properties. This inability to selectively apply properties to classes of objects in Arjuna arises from two sources:

1. The classes defined by Arjuna are restricted to single inheritance thereby limiting the extent to which properties can be independently applied.
2. In some cases, the "property" classes defined by Arjuna incorporate more than one concept in the same class. For example, unique network-wide identity is mixed up with the property of persistence. While this kind of identity is required for persistent objects, not all identifiable objects need be persistent.

2.8.2 Camelot / Avalon

Camelot [Spector *et al* 87] [Spector *et al* 88] is a successor to the **TABS** system [Spector *et al* 85], augmenting the TABS concepts for virtual storage management and recovery with optimised commit protocols as in **R*** [Mohan *et al* 86] and with support for nested transactions. Camelot is based on the Mach operating system and provides a conventional non-object-oriented interface for programming. **Avalon** [Herlihy and Wing 87] is a language layer that embeds Camelot facilities in some host language e.g., C+ +. In this way, powerful concurrency control and recovery facilities are provided to programmers in a syntactically integrated interface. However, since the mechanisms of Camelot are implemented by non-object-oriented means, some of the power of the object-oriented interface is lost. For example, extending the recovery behaviour of objects using inheritance in Avalon/C+ + is complicated because the underlying system features do not have the same semantics of subclassing and inheritance. The lack of object-oriented design also inhibits the ability to develop application classes that inherit multiple properties.

2.8.3 RelaX

RelaX [Kroeger *et al* 90] [Schumann *et al* 89] is a system software interface supporting nested transactions on a wide range of object types. The management of recovery, synchronisation and persistence in RelaX is very flexible, but RelaX is not object-oriented in either its interface or implementation, relying instead on a more traditional imperative programming interface. The result is a system with limited extensibility. The basic system offers many primitives which can be combined in many ways, but extending the set of primitives is difficult because of the lack of object orientation.

2.8.4 ISIS

ISIS [Birman and Joseph 87] [Birman *et al* 88] is a toolkit for distributed programming that represents a non-transaction based approach to the development of reliable distributed applications. Rather than employing transactions to provide fault-tolerance, ISIS depends on a checkpoint facility and a broadcast communications mechanism designed to support a layered set of protocols for ensuring atomicity and causality. The tools of the ISIS toolkit are combined with a concept called "virtual synchrony" which is interesting in its own right. This concept is powerful in application areas where transactions are unsuitable or impractical because backward error recovery is not possible, for example, certain process control applications. However, ISIS has limited applicability to the traditional domains of transaction systems.

2.8.5 Related Systems Summary

These four systems, Arjuna, Camelot/Avalon, RelaX, and ISIS have contributed greatly to the design ideas developed in this thesis. Each system is an example of an attempt to provide a coherent programming environment for the development of reliable distributed applications. Table 2-1 summarises the capabilities of these systems relative to the requirements developed in the previous sections. The most important issue which is not adequately addressed by any of these systems is the selective application of properties to objects or object classes. Although these systems have provided limited separation of properties, for example separating persistence from concurrency control, they have failed to clearly separate all the properties implied by the different forms of transparency – especially recovery and persistence.

System Requirements	Arjuna	Camelot/Avalon	RelaX	ISIS
Object-Oriented Interface	✓	✓	✓	
Object-Oriented Design	✓			
Reliability support	✓	✓	✓	✓
Persistence Transparency	✓	✓	✓	
Distribution Transparency	✓	✓	✓	✓
Selective property support				

Table 2-1: Related Systems Summary Chart

Beyond consideration of these systems for reliable distributed programming, there are numerous related works in the areas of programming languages and distributed operating systems. Section 2.4 has already discussed the contributions of persistent programming languages to the concept of persistence transparency. Other research into programming language and operating systems has contributed in various ways to our understanding of distribution transparencies and object-oriented programming, as described in the next two sections.

2.8.6 Programming Languages

Considerable research has been undertaken in the areas of object-oriented design and programming. Many object-oriented programming languages have been designed and implemented e.g., **Smalltalk** [Goldberg and Robson 83], C++ [Stroustrup 86] [Lippman 89], **Trellis/Owl** [Schaffert *et al* 86], **Simula** [Dahl *et al* 70] [Birtwhistle *et al* 73] and **Eiffel** [Meyer 88]. Beyond mere syntactic differences, there are important differences in the semantics of these languages particularly concerning their encapsulation model, degree of autonomy for objects and "purity" (i.e., the degree to which *all* programming constructs are *objects).* There have also been attempts to augment programming languages with transaction processing constructs to provide a measure of reliable programming, notably in Argus and Emerald.

Argus [Liskov 84] [Liskov 88] is a programming language for distributed systems that provides reliability support through nested transactions and atomic types. Objects which are instances of an "atomic type" are atomically updated when the action that invoked them completes. Argus has two main concepts: A *guardian* which is a logical node of the system and an *action* which is a nested atomic transaction as described above [Liskov and Scheifler 83].

There are three key differences between Argus and the work reported in this thesis:

- Argus is based on Abstract Data Types rather than objects. Argus does not support inheritance.
- In Argus, no separation is made between recovery, persistence and concurrency control. An "atomic type" acquires all of these properties whether or not it needs all of them.
- Argus uses transactions only for remote invocations. Local operations have no recovery or concurrency control possibilities.

Emerald [Black *et al* 87] [Raj *et al* 91] is a general purpose object-based programming language for distributed systems. Developed in parallel with the Eden distributed operating system, Emerald has extensively explored concepts of distribution transparency, including not only location and access transparencies, but also concurrency and migration transparencies. The key differences between Emerald and this work are:

- Emerald objects are active, using monitors to control concurrent accesses.
- Emerald does not support any kind of inheritance.
- Emerald does not attempt to address fault transparency.

Ultimately, the virtual machine on which a programmer develops an application is a combination of the features of the programming language, the operating system and, in limited ways, the hardware architecture. The distinction between programming language run-time support and operating system support is frequently blurred and of little relevance to the programmer. Indeed, many "language features" have been successfully migrated into operating systems where issues of global resource management and access control can be more easily addressed.

2.8.7 Distributed Operating Systems

There has been considerable research in the past ten years in the area of distributed operating systems. As the next chapter bears out, the design of the operating system - its features and computational model - will have important effects on the provision of reliability, distribution and persistence by higher level software. In many cases, recent research into the design of operating systems has incorporated some combination of reliability features, persistence and distribution. The level at which these facilities are provided is an architectural decision that must be taken with respect to the overall goal of a development environment for reliable distributed applications. Numerous research projects in this area such as **Cronus** [Schantz *et al* 86], **Gothic** [Banâtre *et al* 86], **LOCUS** [Walker *et al* 83] [Walker 85] [Moore 82] and **Zeus** [Browne *et al* 83] influenced the work reported in this thesis. The following paragraphs summarise the most relevant research developments and indicate their relation to the topic of this thesis.

Clouds [Allchin 83] [Allchin and McKendry 83][Dasgupta *et al* 88] [Pitts 88] is a distributed operating system developed at the Georgia Institute of Technology with a specific goal of integrated reliability. Clouds supports reliability by offering nested transactions, including replicated transactions. Location transparency is supported through the provision of a global object name space. Access transparency is provided by transparent remote invocation. Concurrency transparency and fault transparency are provided by the transaction mechanism. As in Argus, transaction boundaries are determined semi-automatically by causing the transaction commit operation to occur as a side effect of the termination of a "consistency-preserving" thread of control. A thread of control becomes a "consistency-preserving" thread automatically by invoking a "consistency-preserving" operation of an object. In Clouds, all objects are notionally persistent and a distributed garbage collector is employed. Clouds differs from the work reported in this thesis in the following ways:

- Clouds supports objects, but does not provide an object-oriented interface; Aeolus, the Clouds programming language does not support inheritance or subclassing.
- Clouds clearly separates concurrency control from recovery, but fails to separate recovery from persistence.
- Clouds transaction boundaries are automatically determined, restricting the level of control which application programmers can exercise.

Guide [Balter *et al* 91] is an object-oriented distributed operating system supporting reliable computing through nested transactions. As in Argus and Clouds, objects may be attributed with the property "atomic", which implies recovery and concurrency control. All objects are implicitly persistent and a distributed garbage collector is employed. Guide fails to address some issues treated in this thesis as follows:

- In Guide, no separation is made between recovery, persistence and concurrency control. An "atomic type" acquires all of these properties whether or not it needs all of them.
- Guide employs a special programming language to make its atomicity features available and is thus restricted in its applicability.
- Inheritance in Guide is limited to single inheritance and is not employed to provide the properties of recovery and concurrency control.

SOS [Shapiro *et al* 89] is a distributed, object-oriented operating system supporting distribution transparency including location, access and migration transparency. While SOS is not focused on reliability, SOS provides most of the operating system facilities that provide the necessary support environment for distributed transaction support and persistent programming.

Amoeba [Tanenbaum and Mullender 81] is a distributed operating system based on processes, messages and ports, with a specific goal of efficient use of network resources. Its relation to this work is in its use of global object identifiers and distribution transparency, including location and access transparency and migration. The object support in Amoeba derives from its use of capabilities to refer to and provide access control for objects anywhere in the system. Unlike the work in this thesis, Amoeba does not provide any direct support for reliability or availability. Although Amoeba support objects in a primitive sense, it does not employ object-oriented concepts such as inheritance in its interface or implementation.

Eden [Lazowska *et al* 81] is a distributed object-based operating system developed at the University of Washington. In conjunction with the programming language Emerald, described above, Eden provides primitive support for active objects, including migration support. The Eden File System [Jessop *et al* 82] supports nested transactions, although the interface is not object-oriented in the sense that no use is made of inheritance. Eden does not support transactions for general program objects - only for file system objects.

Choices [Campbell *et al* 89] is an object-oriented operating systems development kit. Sometimes referred to as a family of operating systems, Choices provides basic operating systems facilities using an object-oriented design and implementation. While Choices makes no particular provision for reliability, its object-orientation and idea of feature composition and tailoring by the use of object-oriented aggregation and inheritance closely parallels the work of this thesis. The Choices system also provides support for persistent objects and consideration has been given to the general implications of persistence support for operating systems design [Campbell and Madany 91].

Chorus [Rozier *et al* 88] and **Mach** [Jones and Rashid 86] are distributed operating systems based on processes, messages and ports. There is no particular emphasis on reliability in either Chorus or Mach. However, some recent attempts have been made to build fault tolerance mechanisms "on top" of Mach using process pairs and periodic checkpointing rather than the "application level" concept of transactions [Babaoglu 90]. **COOL** [Habert *et al* 90] is a layered extension to the Chorus system to provide facilities for object management including migration. These layered fault-tolerance mechanisms build object-oriented transaction and persistence features on a basic operating system kernel.

System Requirements	Clouds	Guide	SOS	Amoeba	Eden	Choices	Chorus	Mach
Object-Oriented Interface	✓	✓	✓	✓	✓	✓		
Object-Oriented Design						✓		
Reliability support	✓	✓	✓					
Persistence Transparency	✓	✓	✓	✓				
Distribution Transparency	✓	✓	✓	✓	✓	✓	✓	✓
Selective property support						✓		

Table 2-2: Distributed Operating Systems Summary Chart

Table 2-2 summarizes the properties of various distributed operating systems relative to the requirements stated in section 2.5. Note that the systems are treated fairly generously in that all are attributed with distribution transparency although not all systems support all of the transparencies. In particular, Eden and SOS are the only systems on the list that have extensively considered migration of objects. However, as mentioned above in section 2.3.1, migration and replication transparencies are second order problems since they depend on the existence of access and location transparency. Hence, the four primary transparencies, access, location, concurrency and fault transparency, have been used in the construction of this chart.

2.8.8 Client - Server model

The concept of a "client - server architecture" has been widely applied in modern distributed systems. For example, NFS [Sun 88] employs a client-server design, although it is simply a network file system, not a complete operating system or programming environment. In the client-server model of computation, an active process (the client process) on one node invokes an operation provided by a server (process) which may be on a different node. The operation invocation is typically by message passing and usually involves a result which is returned in a subsequent message. Although many variations are possible (e.g., allowing asynchronous reply), the most common style of interaction involves blocking the client until the server has completed the requested operation and responded. This style of synchronous invocation maps easily to conventional programming notions of procedure calls and has led to a proliferation of "remote procedure call" facilities.

The model presented in this thesis adapts the conventional client - server model with the difference that the entities that are communicating are *objects*. The client application (object) invokes an operation provided by a server (object) which may in turn invoke an operation of another server. Thus a single running process may be simultaneously acting as a client and as a server.

2.9 Summary

Many abstractions have been developed to address various aspects of reliability, persistence and distribution of objects. One choice for an integrated set of these abstractions includes nested transactions on persistent objects, employing a remote procedure call communications model augmented with naming and locating support. This chapter has explained the rationale for this choice of abstractions, the mechanisms needed to implement them, and the relationship between this work and prior developments.

In summary, basic distribution transparencies such as access transparency, and location transparency are necessary precursors to more advanced forms of distribution transparency such as replication transparency or migration transparency. Transparent concurrency control and fault semantics are also necessary to achieve a simple uniform programming model. Atomicity implies recovery transparency and some aspects of persistence, for example, to achieve the *permanence of effect* property of transactions. When the concept of persistence transparency is added to the list of transparencies, the necessity for the separation of persistence from recovery becomes clear. The importance of such selective attribution of properties to objects has been emphasized.

The next chapter describes an architecture for reliable distributed systems, explaining the role of support services such as naming and location services and emphasizing the potential for portable systems development by defining clear, narrow interfaces to operating system services.

3 Architectural Issues

"The Machine is the architect's tool – whether he likes it or not. Unless he masters it, the Machine has mastered him. The Machine? What is the Machine? It is a factor Man has created out of his brain, in his own image – to do highly specialized work, mechanically, automatically, tirelessly and cheaper than human beings could do it. Sometimes better."
Frank Lloyd Wright [Wright 1927]

3.1 Introduction

The architecture of a distributed information processing system can have a profound effect on the structure, efficiency, portability and reliability of applications written for it. To minimise the dependence of applications on operating system and machine architecture, it is useful to define the programming interface to these low level services in terms of high level abstractions that provide the distribution and persistence transparencies described in Chapter Two. Attainment of the desired levels of transparency requires the support of various mechanisms; for example, access transparency requires a remote communications mechanism. The combined set of mechanisms for support of persistence, communications, naming and location, recovery, concurrency control and transactions must be organised into some structure, that is, into an architecture that provides a sensible programming interface to the full range of transparency services. This chapter describes one such organisation that minimises dependence on specific operating system features, explains the rationale for its selection and discusses the implications of the architectural alternatives which have been chosen.

3.2 System Architecture

To provide an architectural view of a distributed transaction processing system, the following list summarizes the requirements of each of the desired transparencies together with the system dependencies inherent in the implementation of mechanisms to support those transparencies.

- Access transparency requires an interface for non-local communications which mirrors local communications. In the case of object-oriented programming languages, the communications model could be based either on explicit message-passing, as in Smalltalk, or on operation invocation, as in Eiffel, C++, Trellis/Owl and others. If the programming language employs a message passing model for inter-object communications, the mapping to distributed systems is clear. If inter-object communications are based on synchronous operation invocation, Remote Procedure Call (RPC) seems a natural mechanism to support access transparency. As discussed in Chapter Two, RPC is adopted in this thesis as the work is based on an object-oriented programming language that uses synchronous operation invocation to communicate between objects.

  Communications in the distributed system requires access to communications devices which may vary from small local area networks to wide area networks involving thousands of nodes. The many different devices provide many different interfaces, making the implementation of the communications interface a system dependent problem.

- Replication and migration protocols can be incorporated into the chosen communication mechanism to provide higher levels of performance, reliability or availability. In both the message passing model and the RPC model, replication support will imply that multiple physical messages are sent in response to a single logical communication. Migration involves not only physical moving the object, but also updating the location service in some way to preserve location transparency.

- Location transparency requires a mechanism that can locate any object in the system using a uniform interface, regardless of whether the object is local or remote. In fact, although a single interface is required, several mechanisms will be used to implement that interface to efficiently support location services in different circumstances. Purely local, transient objects may be identified and located by means internal to the programming language implementation, e.g., stack offsets or heap addresses. For shared or persistent objects, some kind of *location service* must be provided. Naming services that map programmer-readable names of objects to some kind of unique, unambiguous identity must also be provided and may be combined with location services.

Note that the mappings maintained by the name service and location service are themselves examples of replicated, shared, persistent objects which should be updated under the control of some concurrency control protocol.

- Persistence transparency must be supported by some underlying mechanism that involves external storage media such as disks, tape, or optical storage devices. Access to these kinds of devices will be dependent on the specific details of the hardware and low-level software interfaces. The persistent storage subsystem may escape dependence on such low-level device interfaces if the operating system on which it is based provides a measure of device independence, but the storage subsystem will still be dependent on the operating system interface.
- Failure transparency, provided by recovery mechanisms and co-ordinated by a transaction protocol, and Concurrency transparency, provided by locking or timestamping mechanisms, can be built using the other facilities of the transaction services kernel without reference to other operating system or hardware facilities. Transactions can be distributed without introducing further operating system or hardware dependencies if access and location transparencies are provided.

From this list of the transparencies and the kinds of support they require, it is clear that the mechanisms necessary to support distribution and persistence transparencies may be loosely divided along the lines of system dependence and independence. Indeed, previous research in this area has addressed the issue explicitly. Some existing environments for applications programming for distributed systems have focussed on heterogeneous RPC facilities [Bershad *et al* 87] and common run-time environments [Weiser *et al* 89]. Such systems support access transparency even in heterogeneous networks, but fail to provide the range of facilities necessary to support reliable distributed transaction processing for persistent objects. Yet, these previous works have demonstrated that only a small kernel set of the mechanisms must interface directly with operating systems or hardware systems to provide the desired services. The whole set of support mechanisms can be layered in such a way as to depend on only two basic kinds of operating system service: multicast communications and stable storage. From these two basic services, higher level services such as naming, location, persistence, and RPC services can be constructed to provide a suitable environment for support of concurrency control, recovery and transaction management.

From these observations, the following key architectural components emerge:

- Communications: To provide an object invocation facility through an RPC mechanism
- Object Storage: To provide a repository for objects in stable storage
- Naming: To provide a mapping from user–given names of objects to system–assigned unique identifiers
- Locating: To provide a mapping from unique identifiers to location information
- Transactions: To provide atomic action support to application programs, including recovery, concurrency control and persistence

The relationship amongst these services is depicted in Figure 3–1.

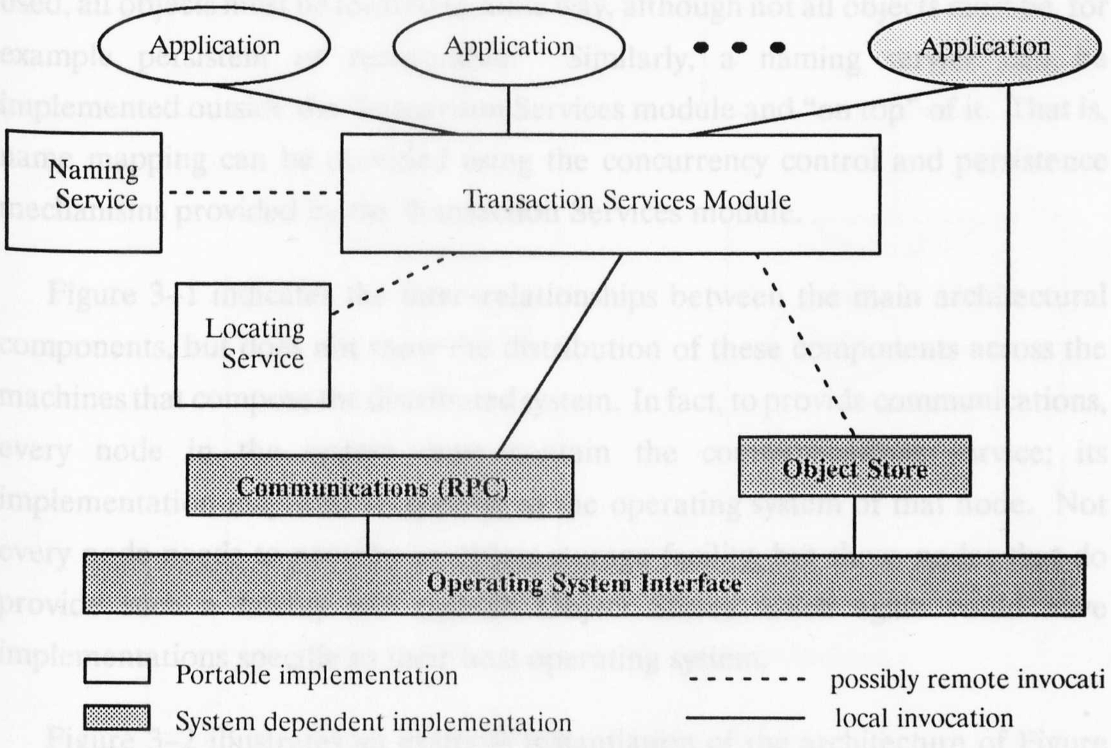


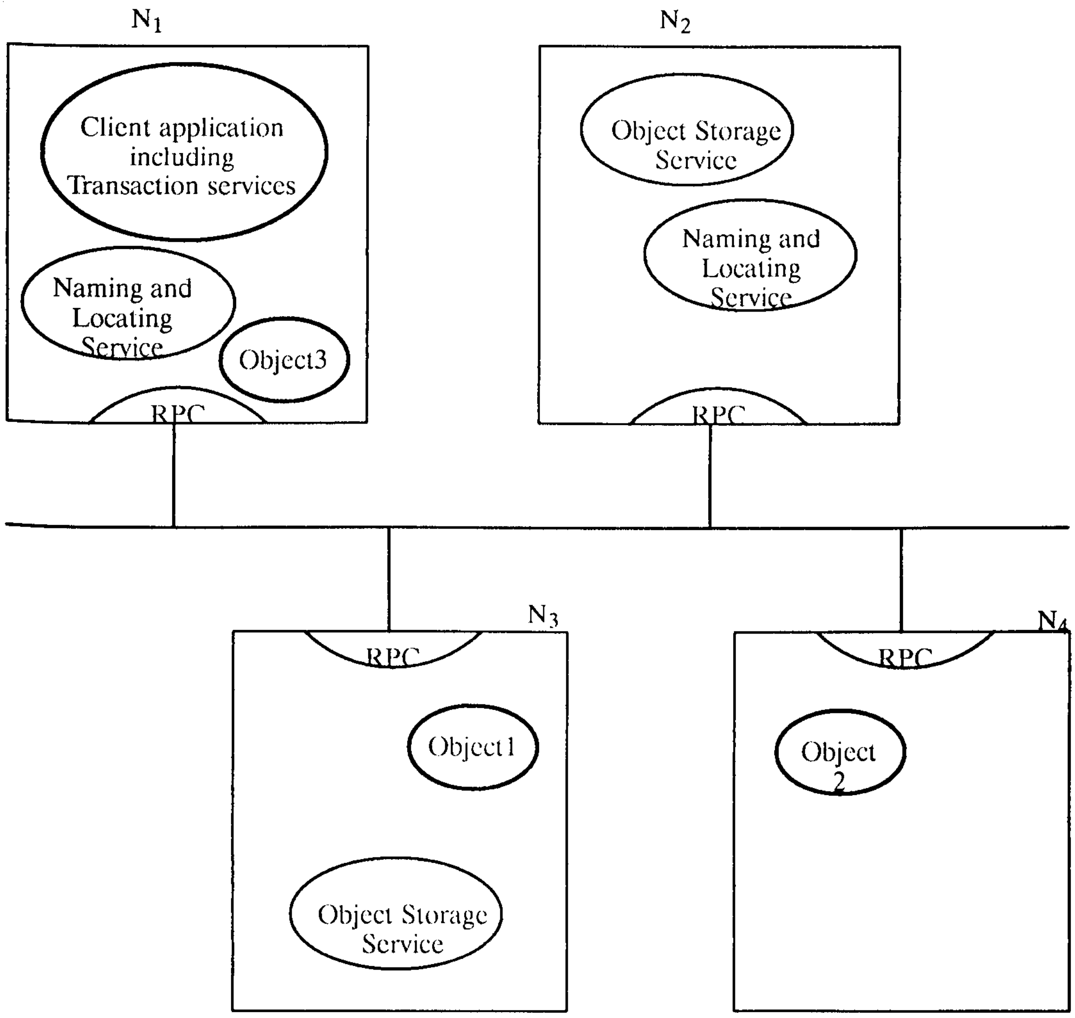
Figure 3–1: Components of a Persistent Object Transaction System

To gain the benefit of the transparencies provided by the system, an application uses objects and operations defined in the Transaction Services module which in turn use the lower–level services. Since the Transaction Services module includes the ability to access local and remote persistent objects, complete applications can be written using only this service without any direct access to other operating system facilities. An application may also directly access the host operating system services (see Figure 3–1). Such an application can maintain portability by using portable sub–systems for all services. For example, an application which requires a graphical interface might use a portable system such as the X Window System [Scheifler and Gettys 86] in addition to the services provided by the Transaction Services module. However, recoverability, availability, persistence, concurrency control and transparent communications will not be provided for the window manager objects. In effect, the Transaction services module provides *portable* persistent object support with the transparency properties described in Chapter Two.

As shown in Figure 3–1, location transparency can be supported by services outside the main transaction system. This architectural decision arises from the assumption that there is no programming construct associated with location; to be used, all objects must be located in some way, although not all objects must be, for example persistent or recoverable. Similarly, a naming service can be implemented outside the Transaction Services module and "on top" of it. That is, name mapping can be provided using the concurrency control and persistence mechanisms provided by the Transaction Services module.

Figure 3–1 indicates the inter–relationships between the main architectural components, but does not show the distribution of these components across the machines that compose the distributed system. In fact, to provide communications, every node in the system must contain the communications service; its implementation may well be specific to the operating system of that node. Not every node needs to provide an object storage facility, but those nodes that do provide such a facility will contain Object Stores which again could have implementations specific to their host operating system.

Figure 3–2 illustrates an example instantiation of the architecture of Figure 3–1 on four nodes to exemplify the relationship of the services to application programs. In this example, an application program containing references to persistent objects is executing at node N_1. This application program is the root of the computation. Distributed execution is achieved by invoking operations on remote objects, *Object*$_1$ and *Object*$_2$ on nodes N_3 and N_4 respectively. The set of nodes involved in a transaction will be determined dynamically according to the objects and operations accessed in the course of executing the application program. In this example, at the stage depicted in Figure 3-2, the application program has referenced objects on nodes N_1, N_3 and N_4. Node N_2 may also be involved in the distributed computation if, for example, Object$_3$ is a persistent object stored in the Object Store at N_2.



N1
Client application including Transaction services
Naming and Locating Service
Object3
RPC
N2
Object Storage Service
Naming and Locating Service
RPC
N3
RPC
Object1
Object Storage Service
N4
RPC
Object 2

Figure 3-2: An Instantiation of the Architecture

In the sections that follow, each of the principal components of the architecture is described. Each description includes a discussion of the role of the component in providing support for distribution and persistence transparencies and a few remarks about implementation issues that arise from architectural decisions. A detailed discussion of implementation issues is deferred to Chapter Five.

As indicated in Figure 3–1, the two architectural components which form the interface between the host operating system and the distributed transaction processing system are the communications module and the object storage module. These components are discussed first, followed by the naming and locating modules and finally, the programmer's interface to the entire system, the transaction services module.

3.2.1 Communications (RPC)

The purpose of the RPC service is to provide access transparency for distributed applications. In the client–server model adopted here, the application is the initial client and the objects are the servers. The operations of an object may make use of other objects, in which context the first object becomes a client and the invoked object is the server.

To transparently interface the synchronous operation invocations of the application program to possibly remote objects, a suitable stub generator can be used to pre–process the class descriptions to provide the RPC stubs necessary to interface to the lower level communications interface. The details of such RPC stub generating systems are well explained in the literature [Bershad *et al* 87] [Gibbons 87] [Parrington 90].

The RPC service provides three operations: *call*, to be used by the client, and *get_request* and *send_reply*, to be used by the server. Of course, appropriate operations for establishing and breaking connections must also be provided. Clients and servers have communication identifiers (CIDs), such as sockets in Unix, for sending and receiving messages. The architecture is equally valid for RPC systems employing synchronous (get–request, send–reply) interfaces or asynchronous (action routine registration) interfaces. The RPC system is designed to cope with conventional network faults (e.g., duplicate messages, lost messages).

3.2.2 Object Storage

To provide persistence transparency, the Transaction Services module employs operations of Object Stores to access the saved state of objects. Persistent objects not in use are said to be in a *passive* state. A passive object is made *active* by loading its state and methods from an Object Store to the volatile store. A persistent object must provide operations to enable it to be activated or de-activated. Note that this operational description of the activation and de-activation of objects is intended to give the semantics of object activation rather than a precise description of the mechanics. An implementation of the object store is free to use caching, pre-fetching, clustering, etc. to optimise the movement of objects (code and data) from stable storage to volatile storage and back. Furthermore, each object is responsible for providing concurrency control operations through mechanisms such as locking or path expressions describing permissible concurrency within an object [Campbell and Habermann 74]. An essential point is that programmers must be able to treat persistent objects in the same way that they treat local objects. Like local objects, persistent objects are "active" as soon as they are in scope – an extra "activation" step must not be required for persistent objects as this would violate transparency.

Like the RPC service, the Object Store has two interfaces: a client interface – available to every application, and a server interface present only on the nodes containing object stores. The client interface, with support from the RPC interface, hides the potential remoteness of object stores from the applications, while the server interface of an object store hides the system specific details of stable storage.

The passive representation of an object may differ from its active volatile store representation. For example, an object in volatile store may contains pointers to other objects as part of its state. When the object is to be saved in an object store, the pointers in the volatile state of the object may be represented as offsets or UIDs in the passive state of the object. The passive representations of objects are instances of the class ObjectState. A persistent object is assumed to be capable of *packing* its state into an ObjectState instance and *unpacking* a previously packed ObjectState instance into its instance variables by invoking operations on ObjectState objects. These packing and unpacking operations will be invoked (automatically by the Transaction Services module) to make transitions between active and passive states of objects.

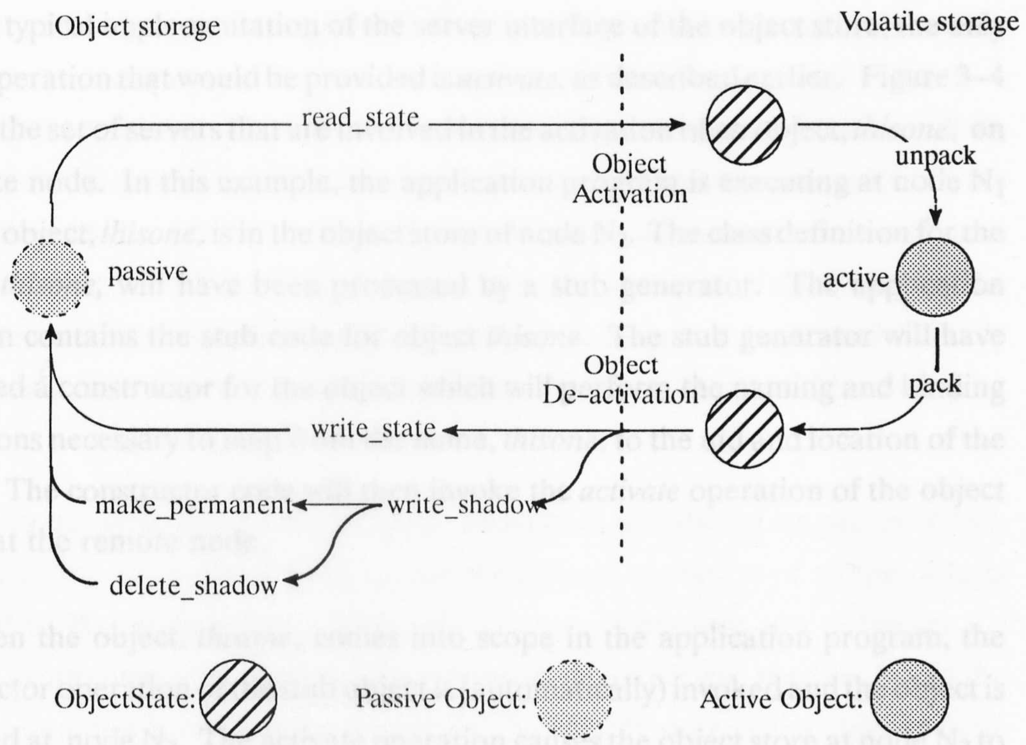


Figure 3–3: Object State Transitions

Figure 3–3 shows the lifetime and state transitions of a persistent object along with the operations that produce the transitions. Operations to the left of the vertical dashed line e.g., *read_state, write_state* are performed by the Object Store, the other operations are performed by the Transaction services module. The class ObjectState ensures that the instance variables of an object are stored in a form that may be transmitted between nodes with different architectures. As a result, instances of the class ObjectState may be sent in messages to other nodes to support object caching, migration and recovery.

The primary function of an object store is to store and retrieve instances of the class ObjectState: the *read_state* operation returns the instance of ObjectState named by a uid and the *write_state* operation stores an instance of ObjectState in the object store under the given uid. The operations, *write_shadow, delete shadow* and *make–permanent* are provided to support a recovery technique based on a shadowing mechanism. The requirements of various recovery techniques and the implications for the interface to the ObjectStore will be discussed further in Chapter Four, but for the rest of this chapter, we will assume that recovery is based on a shadowing mechanism. Two additional operations (not shown) are also necessary: *create* and *delete* for creating and deleting objects.

In a typical implementation of the server interface of the object store, the only client operation that would be provided is *activate*, as described earlier. Figure 3–4 depicts the set of servers that are involved in the activation of an object, *thisone*, on a remote node. In this example, the application program is executing at node N_1 and the object, *thisone*, is in the object store of node N_2. The class definition for the object, *thisone*, will have been processed by a stub generator. The application program contains the stub code for object *thisone*. The stub generator will have produced a constructor for the object which will perform the naming and binding operations necessary to map from the name, *thisone*, to the uid and location of the object. The constructor code will then invoke the *activate* operation of the object stored at the remote node.

When the object, *thisone*, comes into scope in the application program, the constructor operation of the stub object is (automatically) invoked and the object is activated at node N_2. The activate operation causes the object store at node N_2 to create a server (thread or process) to manage the active state of the object *thisone*. Using the uid of the object, *thisone*, the server retrieves the ObjectState instance which contains the passive state of the object *thisone* from the stable store, loads the methods of *thisone's* class into the server process, and invokes the *unpack* operation to restore the active state of the object *thisone* in volatile memory. All subsequent operation invocations on the object *thisone*, including concurrent invocations by other transactions, will use the RPC mechanism to access the server on node N_2 which is servicing the object, *thisone*.

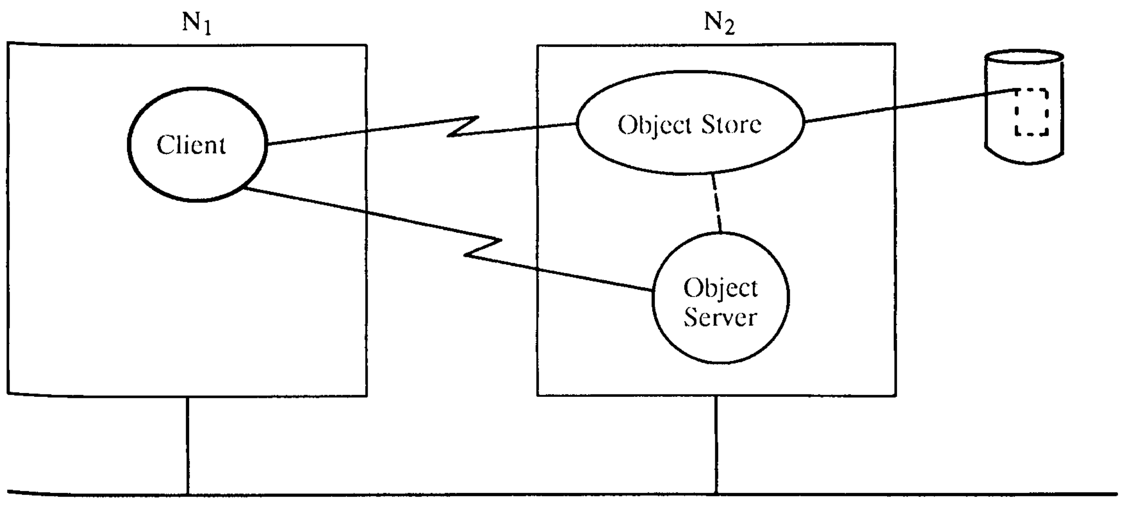


Figure 3–4: Objects, Clients and Servers

Application programs or servers that create transactions will act as the transaction co-ordinators for the transaction which they initiated. When a transaction completes, the decision to commit or abort that transaction will be made by the co-ordinator after communications with the objects involved in the transaction. Although the decision is made in one place, the co-ordinator must first communicate with the objects (this is, with the servers that are managing the active state of those objects) because the object state and all associated recovery state, persistence state and concurrency-control state are held in the servers.

At commit time, using a conventional two-phase distributed commit protocol, the transaction co-ordinator will invoke the *prepare* operation of all the objects involved in that transaction. Three object store operations are necessary for commit processing: *write_shadow, make_permanent,* and *delete_shadow.* When the *prepare* operation is received by the server, the volatile state of the object will be *packed* into an ObjectState and the object store operation, *write_shadow,* will be invoked to create a stable, passive version. If the server subsequently receives a commit invocation, it executes the *make_permanent* operation of the object store to convert the temporary version into the new stable state of the object. The response of the server to an *abort* operation is to execute the *delete_ shadow* operation and to discard the volatile copy of the object.

To summarize: an object store supporting recovery by shadowing provides seven operations (*create, delete, read_state, write_state, write_shadow, delete_shadow, make_permanent*) which are used by a persistent object, which itself exports operations *prepare, commit* and *abort*. A client program need only use the *activate* operation of an object store before accessing the object.

It should be noted that a transaction itself needs to record some recovery data on stable storage: an intentions list [Bernstein *et al* 87] for committing or aborting the action in the presence of node failures. An intentions list is a kind of persistent object associated with a transaction. The next chapter describes the way in which the basic facilities of the transaction services module are employed to manage the intentions list data associated with transactions.

3.2.3 Naming and Locating Services

Naming services are required to locate objects by name and to manage naming contexts. Such services are often designed together as a part of a "name server" which becomes responsible for mapping user supplied names of objects to their locations [Oppen and Dalal 81]; sometimes such services are implemented as an integral part of the RPC system [Black *et al* 87]. However, naming and location services can be looked upon as logically different subsystems, related to applications as indicated in Figure 3–1.

The human-readable names associated with objects are a convenience for application programmers, not a fundamental part of the system's operation; the Transaction services module is responsible for manufacturing an object name (uid) and assigning it to an object. The mapping from names of persistent objects to their corresponding UIDs is performed by the Naming service operation, *lookup*. The Naming service can be implemented entirely as an application using the Transaction Services. The apparent recursion in design in which the Naming Services use the Transaction Services and the Transaction Services use the Naming Service is easily broken by using well-known identifiers for accessing the Naming Service. In addition to the *lookup* operation, the Naming service must also provide *add* and *delete* operations for inserting and removing names in a given naming context.

Considerable research effort has been invested in designing flexible, efficient naming services. For example, the International standard, ISO, X.500 directory service, the ANSA Trader [ANSA 89], Sun NIS [Sun 88] and OSF's DNS are widely-available standard services. The Naming Service described in this architecture could be implemented using one of these existing services rather than depending solely on the Transaction services for persistent object storage. However, the portability of the system then becomes dependent on the portability of that additional service.

The Locating service, which maps UIDs to nodes, can also be designed as an application. In addition to the *locate* operation, *add* and *delete* operations must also be made available. For performance reasons, locating services are often implemented with special purpose protocols bypassing the RPC service and/or are integrated with Naming Service.

3.2.4 Transaction Services module

The transaction services, including transparent access to objects, remote or local, transient or persistent, can be designed in a wide variety of ways with different tradeoffs. One distinguishing characteristic of the design of the transaction services is the extent to which it is language-specific. This decision affects other aspects of the design profoundly. To provide a language-independent service, a common approach involves designing a module providing language independent primitive operations such as begin-action(), end-action(), and so forth which can be used by arbitrary application programs. Language-specific interfaces to such a "subroutine library" can be developed and multi-language applications can be constructed. (c.f. Camelot/Avalon).

If a language-specific approach is taken, the next decision that must be made is whether to construct a support library using the structuring mechanisms of an existing language or to develop a new language, either from scratch or by extension of an existing language. One alleged benefit of object-oriented languages is in the ability to construct powerful user-defined abstractions which merge seamlessly and elegantly with the primitives of the base language. To the extent that this is true, it should not be necessary to design a new language to provide these mechanisms. Furthermore, the provision of inheritance mechanisms in object-oriented languages provides scope for application-specific enhancements to the basic services, such as type-specific concurrency control or type-specific recovery methods, which are difficult to provide in the language-independent approach. The usefulness of this extensibility and the means by which it is achieved are discussed in more detail in Chapter Five.

3.2.5 An Example

To illustrate the interactions between the various components of the architecture, Figure 3-5 contains a small fragment of C++ code which performs an operation on a single persistent object from within a transaction. The following line-by-line account explains the actions that occur as the example program executes.

```
1.   Transaction A;
2.   Example B ("thisone");

     // start of atomic action A
3.   A.Begin();

     // invocation of operation, OP, on object B
4.   B.op();

5.   if  (...) A.Abort();   // abort the transaction
6.   else A.Commit();       // or commit the transactio
```

Figure 3-5: An Example Transaction

Line 1: An instance, A of class Transaction is created.

Line 2: An instance, B, of class Example is created. The string "thisone" is used at object creation time to access the permanent object by that name. As B is created, the following functions are performed:

- the *lookup* operation of the Naming service is invoked, passing the string "thisone" to obtain the uid of the object;
- the *locate* operation of the Locating Service is invoked to find the Object Store containing the object state of *thisone*; and finally
- the *activate* operation of the Object Store is invoked at the host identified by the location service; the *activate* operation is responsible for activating the object and returning a communication identifier (CID) suitable for RPC communications.

Line 3: A's *begin* operation is called to start the transaction.

Line 4: The operation B.op(...) is invoked. The stub version of this operation which is linked with the application causes an RPC call to be made using the CID established at line 2. This invocation will succeed as long as there are no failures and there are no conflicting invocations on B already in progress. If the invocation is refused due to a conflict, then either it could be re-tried or the entire transaction *A* could be aborted.

Line 5: The action may be *Aborted* under program control, undoing all the changes to B.

Line 6: The *Commit* operation is responsible for committing the atomic action. This is achieved by invoking the *prepare* operation of B (during phase one) to enable B to be made stable. If the *prepare* succeeds, the *commit* operation of

B is invoked (during phase two) otherwise the *abort* operation of B is invoked as the action aborts. These three operations are provided by B especially for transaction termination.

This example illustrates the kind of interface a programmer might use to access the facilities of the transaction services module. The objects in the example are accessed uniformly without regard to their location or other properties.

3.2.6 Replication and Migration Transparencies

The architecture discussed so far possesses the functionality to support all the forms of transparency described in Chapter Two except for replication and migration transparency. These facilities build upon the previous services, typically augmenting the communications service to accommodate multicast communications with specialised protocols for consistency maintenance.

A considerable body of knowledge has been developed on the application of data replication techniques for transaction systems. Such techniques can be adapted to support replica transparency for objects. For example, the available copies approach to replica consistency maintenance - where updates are performed on all available copies of data - can be adapted for replicated objects as follows: the name lookup would return a list of UIDs of object replicas; all of these replicas can be activated by the client performing object activation. Similarly, an object invocation could be converted to invocation of all the activated replicas by the client stub. Object replicas that are found to be unavailable can be excluded from the list maintained at the Naming service. Application level programs can be developed for excluding and joining object replicas. Such techniques can be used to provide the one copy serializability property ensuring that replicas appear to behave like a single object thus achieving replication transparency [Bernstein *et al* 87]. To support object migration, it is necessary is to:

- transmit the state of the object to the destination in a passive form;
- unpack the contents of the received message into the state variables of a new object;
- associate the necessary methods of the object with this state; and
- update location bindings.

One design for a migration mechanism would involve enhancing the functionality of the *activate* operation, supplying it with the identity of the destination host as well as the object UID, thus permitting the object server to be created away from the object store. Initial placement of objects is a useful but limited form of object migration.

A more dynamic mechanism could move an object that was already active, possibly involved in ongoing transactions. Consider the following example. A client has invoked some operation *op* on an object *z* which happens to be remote. The operation z.op requires a single parameter, another object *x* of type *Xtype*. The server executing the operation *z.op* needs to invoke an operation of object *x*; for this purpose the client has passed the object *x* to the server as a part of the RPC message. The server will either have to recreate *x* (if *x* is passed by value), assuming it can determine the type of *x*, or refer back to the client's copy of *x* (if *x* is passed by reference). Often the type information necessary for object creation is not easy to obtain - the situation becomes complicated when *Xtype* has subclasses; say *Xtype* has a subclass *Ytype*, then the server has to determine whether the actual object received as a parameter is an instance of *Xtype* or *Ytype* (inheritance rules permit *x* to be either). Although solutions to this problem can be found, the general observation is that the existence of a class hierarchy, and in particular the use of inheritance, can complicate the mechanization of object migration schemes. These problems have not arisen as a consequence of the architecture presented here, rather they are inherently difficult to solve because of polymorphism in the programming language.

Two further issues concerning object migration must be resolved: concurrency control and updating location bindings. An object is responsible for enforcing its own concurrency control policy - this can to a large extent solve the problem of migrating concurrency control information with the object, since the "concurrency controller" of the object will move with the object. The subject of updating location bindings is somewhat more involved. Suppose that a client wants to access an object *s*. This object *s* is normally resident at node N_i, but has migrated and is currently active at N_j. A simple way of making migration information available to other clients is to leave a "forwarding address" at N_i so that any access to *s* at N_i is automatically forwarded to N_j. This scheme works satisfactorily if objects have "home sites" where they are normally resident, although chains of forwarding can become bottlenecks to both performance and reliability. More complex location finding mechanisms are required if node failures are considered. The Emerald system [Black *et al* 87] attempted to solve the location problem for migrating object using a combination of forwarding addresses and broadcasting. This combination worked tolerably well for Emerald but does not scale well to wide area networks due to the reliance on broadcast communications [Barak and Kornatzky 87]. A more robust solution involving update to the distributed naming service must be employed for better fault-tolerance. It is clear from these comments that if migration of objects is a goal of the design of a distributed object system, the implications of migration must be considered in the design of the location service.

3.3 Support Service Interfaces

Given the architectural framework described in this section, it is now possible to describe the detailed interfaces to the various services necessary for a complete system. The remaining sections of this chapter describe details of the operations provided by the various components identified in the architectural discussion.

3.3.1 Naming Interface

- Lookup (string) -> UID
  Maps the given string to a UID using some explicit mapping which was entered previously using Add. The structure of the string may interpreted by the naming service to define a naming context.
- Add (string, UID)
  Adds an explicit mapping from the string to the UID.
- Delete (string, UID)
  Removes an existing mapping from the given string to the UID.

3.3.2 Locating Interface

- Locate (UID) -> Host
  Returns the host (name, address, or other identity) which contains the object store in which the object designated by UID is stored.
- Enter (host, UID)
  Enters/replaces a location mapping from UID to host.

- Delete (host, UID)
  Removes a mapping from UID to host.

3.3.3 Group Communications (RPC) interface

Client Interface:

- call (CID, ...)
  Sends a message to the server process indicated by the communications identifier, CID, and awaits a reply.

Server Interface:

- get_request (CID, ...)
  Blocks until a message is received on the channel indicated by CID, then returns the message.
- put_reply (CID, ...)
  Sends a reply message to the client whose request was most recently dequeued.

As stated above, there must also be operations to establish and break connections which are available to both client and server.

Communications Quality of Service Requirements

When an RPC *call* operation is invoked, what quality of service is required to meet the demands of the transaction services module? Using well known protocols such as TCP/IP [Leiner *et al* 85] [Postel 81a, 81b] or ISO OSI protocol stacks [ISO7498] [ISO8073], primitive unreliable messaging systems can be made reliable. For simple client-server interactions without hard real-time requirements, a reliable RPC service built upon reliable datagram service will suffice. In particular, session management can be quite limited without adversely affecting the performance or utility of the RPC service. However, when replicated objects are actively involved as servers, multicast communications will be required (to reach all of the replicas) and more stringent reliability and ordering constraints may be imposed. In addition, if migration of active objects is considered, up-to-date location information may need to be acquired before an invocation i.e., RPC *call*, can be successfully completed.

3.3.4 Object Storage Interface

The implementation of the persistence for objects requires an object storage system capable of providing stable storage functions. In the client–server environment in which the transaction management system runs, there are two interfaces to the object store: one as seen from the client, and one as seen from the server. These interfaces are detailed below.

The Client Interface

On the client side, the persistent object class must take the following actions:

- When an object first comes into scope in the application program, a connection must be established to a server which can service operation invocations for this object.
- When the destructor for an object is invoked, the client can discard the connection to the object server.
- All other object operations are passed directly to the server by the RPC mechanism.

To provide this behaviour, the following operations are required in the client interface to the persistent object class:

- Activate (UID, location) -> CID
  The Activate operation takes as input parameters the UID of an object to be activated, and, optionally, the location at which the server should execute. It is up to the implementation of Activate to locate the object which may be at any site in the network. The result of the operation is a communications identifier. The form of this identifier is dependent on the particular RPC interface chosen.
- DeActivate()
  The DeActivate operation destroys the connection to the server.

The Server Interface

Persistent object operations, including constructor operations, are executed inside the server for the persistent object. The server is initially created when an application program executes the constructor for the stub object corresponding to some persistent object (see the operation *activate* above). After the server process is created, the constructor in the server must actually load the state of the persistent object into memory to prepare for future operation invocations. The actions taken by the server in response to the operations of the persistent object class are detailed below.

There must be at least two forms of constructor for persistent objects: one for constructing a memory image of an existing object, the other for creating a new instance. Both forms "construct" an object in memory. The form for existing objects restores its state from the passive image in the object store. The form for new objects simply initializes the state of the new object. Both constructors also record that the object is part of the current transaction (if any) so that appropriate action can be taken when the action commits or aborts.

When a transaction commits, the persistent object operation, *Prepare*, is called first according to the two-phase commit protocol. The persistent object must write its state *provisionally* to the object store. If the prepare phase is successful, the Commit operation is called and the provisional new version of the object must atomically replace the old version.

If the transaction aborts at any stage, the persistent object operation,*Abort*, will be invoked. This operation must inform the object store that the provisional version of the object, if any, is not valid.

Finally, the operation, UnCatalog, removes an object from the permanent store.

From these operation requirements, we derive the following interface for the object store:

- ReadState (UID) -> ObjectState
  The ReadState operation takes as input a UID specifying the object to be read. The result of the operation is an ObjectState object suitable for use in a RestoreState operation.
- ProvisionalWrite (ObjectState)
  The ProvisionalWrite operation copies the ObjectState to stable storage.
- CommitState( )
  The Commit operation causes a provisional version of the object to replace the old version, if any.

- Create (UID, ObjectState, <Location>)
  The Create operation Stores a new object in the permanent store with the identity given by UID. The (optional) location information indicates the host machine at which the object will be stored. The default location will be selected by the object storage system (usually the 'nearest' host).
- Remove ( )
  The Remove operation deletes an object from the permanent store.

Client Interface:

- Activate (UID) -> CID
  Creates (or connects to) a server for the object designated by UID and returns the communications identifier for that server.

Server Interface:

- Create (UID, ObjectState)
  Stores a new object on stable storage. The ObjectState may be subsequently retrieved using UID.
- Destroy (UID)
  Removes the ObjectState designated by UID from the object store.
- ReadState (UID) -> ObjectState
  Reads the saved state of the object designated by UID and returns it (as an instance of the class, ObjectState).
- WriteState (UID, ObjectState)
  Over-writes the existing stable state of the object designated by UID.
- WriteShadow (UID, ObjectState)
  Creates a new, temporary version of the object designated by UID on the stable store. The old state is still stored in the object store and subsequent attempts to read the state will get the *old* state.
- DeleteShadow (UID)
  Removes the ObjectState associated with the new, temporary version of the object designated by UID. The *old* state of the object is not affected
- MakePermanent (UID)
  Causes the temporary version of the object designated by UID (previously written by WriteShadow) to replace the *old* state. Subsequent attempts to read the object will return the new state.

3.4 Architectural Summary

The mechanisms supporting the distribution transparencies outlined in Chapter Two can be implemented in a variety of ways. The organisation of the support mechanisms forms the basic architecture of the system. The architecture proposed in this chapter provides the necessary environment for transaction support, persistence and basic communications. The detailed structure of the transaction services component of this architecture, the programming interface to the system, is the subject of the next chapter.

4 A Class Hierarchy for Actions

"I am perfectly convinced that there will come a time when it will be recognized that programming is one of the more difficult branches of applied mathematics because it is also one of the more difficult branches of engineering, and vice versa."
Edsger W. Dijkstra [Dijkstra 1975]

"It happens that programming is a relatively easy craft to learn. Almost anyone with a reasonably orderly mind can become a fairly good programmer with just a little instruction and practice"
Joseph Weizenbaum [Weizenbaum 1976]

4.1 Introduction

To perform object-level transaction processing, an applications programmer must have a means for accessing and manipulating persistent objects through atomic operations (or operation sequences) i.e. transactions. A hierarchy of object classes can be designed to provide a programming interface which collects the behaviour, that is, the operations, into coherent abstractions which support the distribution and persistence transparencies. These abstractions are the programming language manifestation of the transparencies. Higher-level abstractions can be derived by examining common behaviour of the interface abstractions. This chapter explains the development of such a programming interface, starting with basic concepts of recovery, persistence, concurrency control which directly support the distribution and persistence transparencies. The discussion continues with the development of a complete class hierarchy incorporating transaction management and interfacing to the architecture described in Chapter Three. In fact, the resulting class structure is not a strict hierarchy, since it is not rooted in a single class. It forms a directed graph describing a partial order among the classes. For simplicity however, this class structure will be referred to using the conventional term, class hierarchy.

This chapter is divided into five parts followed by a brief concluding section. Section 4.2 describes the evolution of the class hierarchy from the desired properties at the leaves of the hierarchy through the necessary supporting classes to the root classes, forming a complete hierarchy of property classes for the support of distribution and persistence transparencies. Section 4.3 explains the object class, *Transaction* and its relationship to the property classes. The development of this class library, summarised in section 4.4, is interesting in itself as an example of an approach to object-oriented design. Section 4.5 of this chapter explains the detailed interfaces and semantics for operations of the classes outlined in the first part of the chapter. These details provide a complete description of the externally visible behaviour of the classes.

4.2 Object Properties

Object-oriented programming languages provide a means for programmers to express common operations of a collection of types in the form of an abstract class definition. An object-oriented programming language like C+ + [Stroustrup 86] can be extended to include abstract concepts, such as persistence, by defining *classes* that provide these properties. New classes of objects, derived from these *property classes* will inherit the behaviour of their parent classes. There are several basic properties that can be provided to the programmer in this way:

- identity - for shared or persistent objects, a unique identity must be established. Local, transient objects may not need this "unique identity" property.
- naming - naming involves a mapping from strings to unique identities. These name mappings might not be unique: a single unique identity may have several names associated with it, and a single name might designate different objects in different contexts or at different times.
- recovery - the ability to recover the state of the object to the boundary of the smallest enclosing recovery region.
- persistence - objects with the persistence property will retain their state on stable storage even after the program which created them has terminated.

- concurrency control – shared objects, whether persistent or not, recoverable or not, may require concurrency control.

Each of these abstract properties may be used independently of the others and of transactions. The properties can be intermixed as all combinations have plausible applications. However, in the context of a transaction system, the *operations* of these abstract classes must be invoked according to a strict protocol to guarantee the three transaction properties of serialisability, atomicity and permanence of effect.

Each of these basic properties can be represented by an abstract class which conveys some behaviour to the classes of objects which inherit from them. Other distribution transparencies such as replication, migration, access and location must be dealt with at a lower level because they involve changes to the operation invocation mechanism rather than direct manipulations of an object. As mentioned in Chapters Two and Three, access and location transparency are typically provided by extending the programming language through the use of an RPC pre-processor. Replication and migration transparencies are provided by additional mechanisms layered over the basic access and location transparency mechanisms. In the sections that follow, the "inheritable" properties, *identity, naming, recovery, persistence* and *concurrency control*, and their specific behaviours are described.

4.2.1 Object Identity

Object identity is fundamental to sharing and persistence, and hence a consistent view of object identity is critical to the integration of database-style persistence into programming languages. In their excellent paper on the subject of object identity, Khoshafian and Copeland state, "Identity is that property of an object that distinguishes it from all other objects. Most programming and database languages use variable names to distinguish temporary objects, mixing addressability and identity. Most database systems use identifier keys (i.e., attributes which uniquely identify a tuple) to distinguish persistent objects, mixing data value and identity. Both of these approaches compromise identity" [Khoshafian and Copeland 86]. The authors go on to discuss the impact of identity on the semantics of object equality and conclude that, "The most powerful technique for supporting identity is through surrogates [Abrial 74] [Hall *et al* 76]

[Kent 78] [Codd 79]. Surrogates are system-generated, globally unique identifiers, completely independent of any physical location" [Khoshafian and Copeland 86]. This assertion about the usefulness of surrogates applies to the construction of reliable distributed systems: for persistent objects to be retrieved, they must have an identity which is unique in space and time, and independent of the state of the object. For an object to be shared by two or more transactions, the object must have a unique identity which is independent of the state of the object.

A class of surrogates called unique identifiers (UIDs) must be defined. One way to generate identifying numbers which are sure to be unique in a distributed system is to concatenate a timestamp and some kind of unique host-identifier. Thus, each host can produce unique identifiers autonomously without the overhead of consulting other hosts in the system. This method for construction of identifiers will not defeat the condition of complete independence of physical location laid down by Khoshafian and Copeland as long as the host information in the UID is not relied upon to locate objects.

The adoption of globally unique identifiers for objects results in a two-level addressing scheme for objects. Objects which are active in memory will necessarily be addressed using machine addresses (unless the machine supports direct access via UID such as the Recursiv [Harland and Beloff 87] [Harland 88]). Objects which are remote or passive must be addressed by UID since they have no meaningful machine address in the local context. Emerald [Jul *et al* 88] and Amber [Chase *et al* 89] hide the distinction between local and global references by a combination of compiler-inserted code and run-time support. The possibility of hiding this distinction is only available because these systems employ special programming languages for which customised compilers and run-time support can be constructed. In SOS [Shapiro *et al* 89] as in this work, the distinction is made visible to programmers although support tools are provided to make manipulation of the two types of addressing as straightforward as possible.

The class *UID* provides these unique surrogates. All objects in the system which can be identified by a UID can be grouped together into a separate class, *Identified*. Thus, "identified objects" are just those objects that have UIDs.

4.2.2 Object Naming

While system-generated unique identifiers are important as a means of unambiguous identification of objects, they are not especially convenient for programmers to use. Programmers prefer to refer to objects by names like "/shared/census/data/1989" or "General Ledger". Naming objects with human-readable names, typically managed by a *name server*, can be expressed in the class hierarchy as a new class, derived from the class *UID*, called *NamedUID*. An instance of the class NamedUID is an object which represents the mapping from a string name to a UID. Because the class NamedUID is derived from UID, it *is* a UID and can be used in any context in which a UID is required (see Figure 4-1). The implementation of the class NamedUID will use a name server as described in Chapter Three.

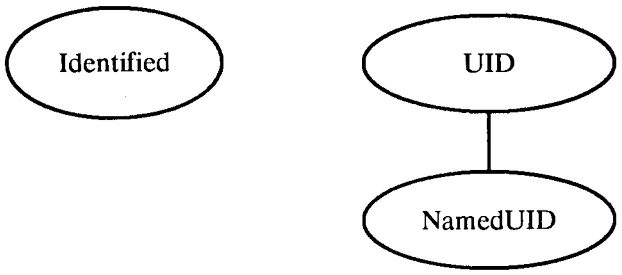


Figure 4-1: Abstract Classes for Object Identity and Naming

4.2.3 Persistence, Recovery, Concurrency Control

The remaining properties desired for programming persistent objects using transactions can be manifest by object classes as follows:

- Persistence can be represented by an abstract class *Persistent,* which provides operations to manage the activation and update of stable (passive) copies of an object i.e., a stable storage interface;
- Recovery can be represented by another abstract class *Recoverable* which provides operations to define the start and end of recovery regions [Lee and Anderson 90] and to manage the restoration of consistent object state in the event of a fault or explicit abort;
- Concurrency control for transaction-level concurrency can be represented by an abstract class *Shared* which provides operations to enforce some concurrency control protocol such as strict two-phase locking or timestamp ordering.

Figure 4–2 illustrates some possible derivations including:

(A) a simple persistent object class,

(B) a temporary recoverable, concurrency–controlled object class, and

(C) a persistent, recoverable, concurrency–controlled object class.

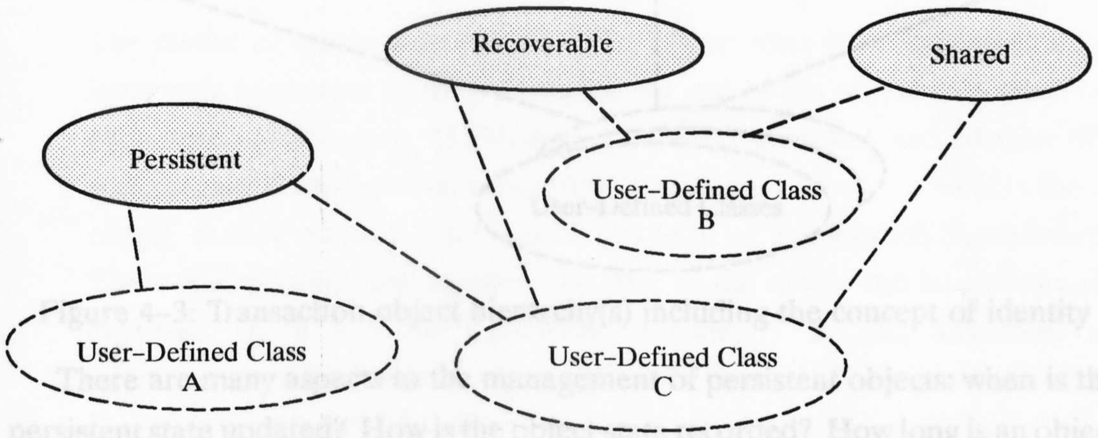


Figure 4–2: Some possible derivations from object hierarchy

4.2.4 Persistence

Creating a class *Persistent* which confers the property of persistence on classes of objects derived from it simplifies a programmer's model of maintaining permanent state. By defining a class of objects that inherits this property, a programmer creates a class of objects that automatically maintain their state across program executions. However, just as files and relations have names that identify the collections that they represent, and elements of files and relations have keys (or seek keys) that identify the individual elements, persistent objects require an identity which can be interpreted across program execution boundaries. Hence, the class *Persistent* must be derived from the class *Identified* (see Figure 4–3).

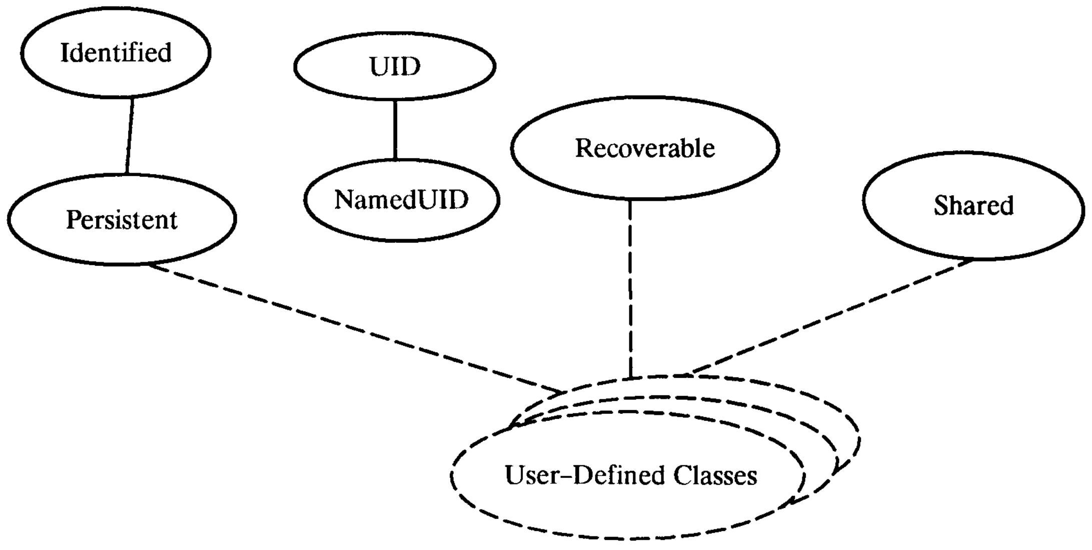


Figure 4–3: Transaction object hierarchy(s) including the concept of identity

There are many aspects to the management of persistent objects: when is the persistent state updated? How is the object state recorded? How long is an object persistent for? That is, are objects explicitly deleted or is deletion automatically managed as with a garbage collector? Some of these questions affect the interface and hence the use of persistent objects; others are only of interest to implementers. Addressing these questions in turn:

- When is persistent state updated?
  This question has a significance in transaction–based systems that may not otherwise be apparent. In a non–transaction system, persistent object state can be viewed as being continuously updated. That is, the permanent state is always the same as the transitory state. When shared access to permanent state is permitted, this definition becomes somewhat ambiguous and when atomic recovery is provided, it may lead to difficulties such as cascaded aborts of transactions that have "viewed" uncommitted changes. A preferred approach for transaction–based system is that the permanent state of an object reflects the last "committed" state of that object. While a transaction is in progress, the permanent state may be undefined. Moreover, the state of an object cannot be assumed to have changed permanently before the end of a transaction, even if the "in–memory" copy of the state has been deleted. The question can be answered simply for a transaction system: the permanent state of an object is updated only upon completion of a transaction.
- How is the object state recorded?
  Since the act of saving (restoring) the state of an object requires knowledge of

the semantics of the object, state capture (restoration) must be explicitly programmed by the class developer. Once the save and restore operations have been programmed, they will be invoked automatically at appropriate times according to the transaction protocol. The interaction of the persistence operations with transactions is discussed fully in section 4.3.1.

The choice of mechanism for recording object state is an implementation issue with important consequences for the use of the object. As observed elsewhere [Parrington 88][Mohan *et al* 89][Rothermel and Mohan 89], concurrency control policies that permit multiple concurrent writers for an object cannot employ state-based recovery or persistence mechanisms. There is simply no time at which the state of the object can be consistently captured vis a vis the concurrent updates. In this situation, an operation-based scheme may be employed. Hence, at least two variants of *Persistent* must be provided:

- *StatePersistent* – a simple state-based variant of the class *Persistent* for which the class implementer must provide operations to save and restore the entire state of the object. This class is suitable for use with objects that use an exclusive writer policy for concurrency control.

- *LogPersistent* – a log-based variant of *Persistent* for which the class implementer must provide "undo–" and "redo–" operations associated with every operation exported by the class. This class is suitable for concurrent writers.

- When / how are objects deleted?
  This important issue has only a minor impact on the interface for persistent objects: the availability of an explicit delete operation. However, the choice of implicit (garbage collected) vs. explicit deletion has a major impact on the use and usefulness of the system. The management of a global heap of persistent, shared objects in a distributed system is a topic of intense research interest at present as evidenced by the wealth of current research in this area cf., [Atkinson *et al* 88] [Bailey 89] [Brown 87, 89]. This is a global issue requiring an architectural solution like the issues discussed in Chapter Three.

The provision of an abstract class for persistence provides a clear, simple mechanism for programmers to express this property and to control the behaviour of save and restore operations. The discussions that follow include both state–based and operation–based techniques. At one level of abstraction, the choice of technique used to capture the permanent state is unimportant. However, the choice must still be expressed and indeed, different "non–functional" properties accrue to the two techniques (as discussed more fully in the next chapter). It is a feature of the inheritance mechanism provided by object–oriented programming languages that the abstract property of persistence can be expressed as a class and applications can operate on objects at this level of abstraction. At the same time, the specific techniques employed to maintain the state, such as, "before–image" snapshots or operation logging, can also be expressed as classes and can be related to the abstract concept of persistence through inheritance. The abstract class *Persistent* conveys the behaviour that the permanent state of an object is updated at the successful completion of a (top–level) transaction, or, in the event of a transaction abort, reverts to the state it had at the start of the transaction. The mechanism by which this behaviour is achieved is determined by the subclass of *Persistent*, in this case, either *StatePersistent* or *LogPersistent*. Unfortunately, these names are somewhat misleading since the aim in both cases is to cause the <u>state</u> of the object to be persistent. The two sub–classes are discussed in turn in the sections that follow.

4.2.5 State–Based Persistence Mechanisms

Since the state of an object will be maintained on permanent storage independent of the execution context in which it was created, a mechanism must be provided to convert an active object into a passive state (e.g., a bit–stream). There are several aspects to this conversion:

- Since the passive state of the object will be re–activated in a different execution context, memory pointers and other context dependent data must be converted to a context independent form (e.g., UID).
- Since the passive state of the object may be re–activated on a machine of a different architecture, the state should be saved in an architecture neutral form analogous to "network byte–order"[Sun 88] and possibly including word size and floating–point number representation encoding, depending on the range of architectures involved.

To encapsulate the semantics of an object whose state can be converted back and forth from a passive, context independent representation to an active, "in-memory" representation, a new abstract class, *CheckpointObject* is defined. The implications of deep-copying vs. shallow-copying in the process of state capture and the related issues of object identity and equality are explored in depth in [Sollins 79]. While automated techniques for "passivating" an object which take account of sharing semantics have been proposed, no robust, general technique has been validated which does not require that programmers be involved in the translation of abstract objects to a passive form. In an object-oriented system, this translation would take the form of object operations for saving and restoration of state. Hence, in the system described in this thesis, the capture and restoration of the state of an object are explicitly programmed by the class developer for each class of objects derived (directly or indirectly) from the class *CheckpointObject.* An additional class is required to represent the passive state of these objects, *ObjectState.* Members of class *CheckpointObject* can save their current state in an instance of class *ObjectState* or restore their state from an instance of class *ObjectState* (see Figure 4-4).

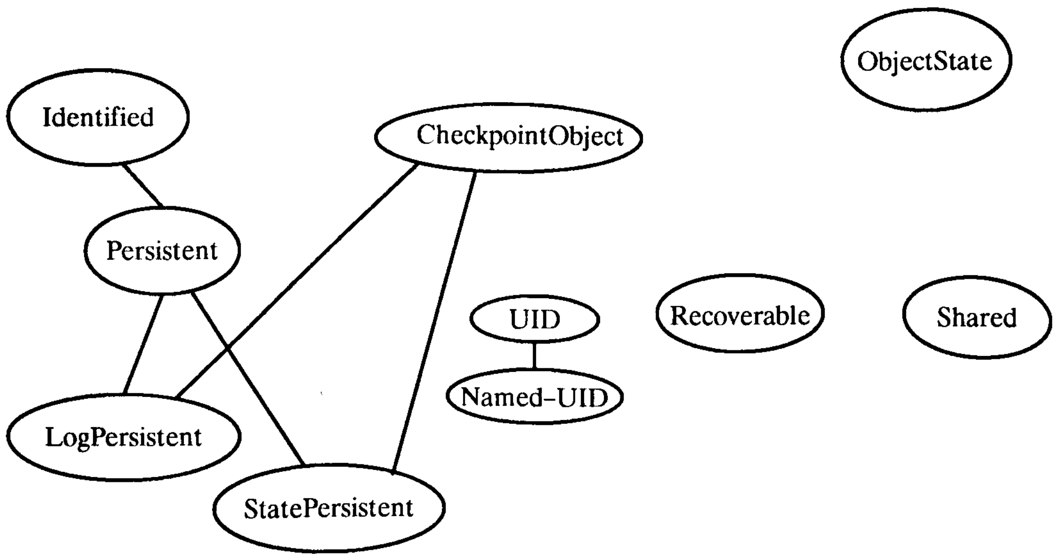


Figure 4-4: An object hierarchy including variants of *Persistent*

The class *StatePersistent* provides the property of persistence to any other class derived from it. However, inheriting from the class *StatePersistent* imposes obligations on the sub-class implementer as well as providing benefits. Since *StatePersistent* inherits from *CheckPointObject*, the implication is that the state of any object derived from *StatePersistent* must be "checkpointable". That is, the operations *SaveState and RestoreState* must be provided by the sub-class implementer. Furthermore, for *StatePersistent* to "know" whether to update the persistent state in stable storage, the sub-class implementer must provide an indication of whether or not the object state has been modified in the course of execution of an application (or some portion of the application). In some object-oriented programming languages, there is an explicit syntax for indicating which operations modify the state of an object and which operations are read-only (e.g., **const** in C++). In other languages there is no such explicit syntax. In any case, it is a small matter for the implementer of a sub-class to add an extra operation invocation to notify the superclass *StatePersistent* whenever the state is about to be modified.

4.2.6 Log-Based Persistence Mechanisms

A variant on simple state capture for persistence, called Write-Ahead Logging (WAL) [Moss 87] [Rothermel and Mohan 89] [Mohan *et al* 89], employs an auxiliary data structure, the log, to improve flexibility and performance of persistent object management. As with the simpler *StatePersistent* class, *LogPersistent* objects must also provide a means for state capture in some architecturally neutral form, that is, they must be *CheckPointObjects*. However, in WAL schemes, operations are logged as they occur and it is only the operation log which must be synchronously flushed to stable storage at transaction commit time. The actual state of the object can be moved from volatile to stable storage as time permits. If a node crash occurs before the state is properly moved to stable storage, the correct, consistent state of the object can be deduced by interpreting the log entries. This ability to delay writing the object state to disk can yield substantial performance benefits over the simple state capture scheme. Moreover, since the log includes information about which transactions have committed and which uncommitted operations have been performed, log-based persistence schemes can be used for objects which have multiple concurrent writers. Hence, a wealth of new concurrency control schemes can be employed when log-based persistence is provided.

4.2.7 Recovery

The ability to recover from an erroneous state is, like persistence, an abstract concept that can be realised in many ways. The definition of the *Recoverable* class is formulated to encompass various recovery techniques (e.g., state-based recovery, operation logging) without modification (i.e., these would simply be different implementations of the interface). Its operation definitions refer only to *establishing* a recovery region, *reverting* to the state at the start of a recovery region, and *discarding* a recovery region [Lee and Anderson 90]. In the context of transactions, these operations would be invoked at action begin, abort and commit respectively. As with the class *Persistent*, the class *Recoverable* has several subclasses each of which employs a particular technique to provide the abstract behaviour of *recovery*. Again, it is possible for many users of *Recoverable* objects to operate entirely at this level of abstraction without ever knowing which technique was employed for which objects. The sub-classes of *Recoverable* which are discussed in the next sections include:

- a state-based recovery class *StateRecoverable*
- an abstract class *OpRecoverable*, covering all operation-based techniques (the existence of this class, grouping all of the operation-based recovery schemes together, is an implementation consideration which is discussed below)
- three operation-based recovery classes derived from the class *OpRecoverable*: *UndoRecoverable*, *RedoRecoverable*, and *CompRecoverable*.

4.2.8 Saving and Restoring Object State for Recovery

For state-based (backward) error recovery there is a requirement to capture the state of an active object at various points in its execution. The same checkpointing operations required to support the persistence mechanism can be used to save and restore the state of the object for state-based recovery. This relationship is explicit in the class hierarchy by the fact that the class *StateRecoverable* inherits from both *Recoverable* and *CheckPointObject* (see Figure 4-5).

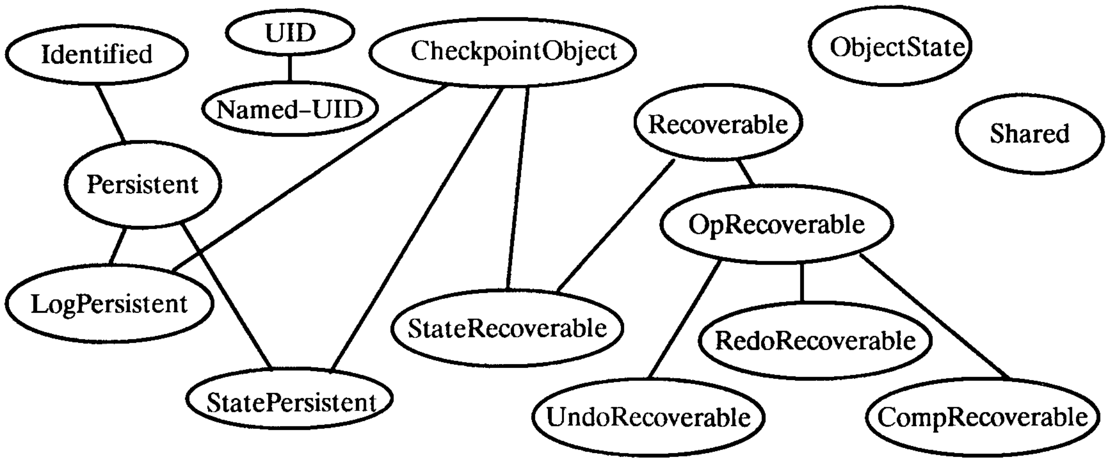


Figure 4–5: Class hierarchy including variants of *Recoverable*

Because *StateRecoverable* employs the *CheckPointObject* operations *SaveState* and *RestoreState* to "programmatically" capture the state of an object, the actual state that gets saved is determined by each class implementer. A large overhead may be associated with capture of the "before image" for a large object. A potentially faster means of capturing the state has been proposed: by augmenting the operations of the virtual memory system of the host operating system, it may be possible to "snapshot" the current memory state of an active object [Traiger 82]. This method, while faster, does make the recovery state dependent on both machine architecture and process context. Migrating an object whose recovery state is so captured could be extremely difficult. There are also potentially large recovery times required when different objects share the same page. However, if migration of active objects is not considered and page placement issues are resolved, the shadow paging technique may well be the most efficient way to effect state–based recovery.

The operation–based recovery classes share many implementation requirements such as the ability to capture a sequence of operation records and to process them in order at appropriate phase changes in transactions. These common implementation requirements are captured in the class *OpRecoverable.*

The class *UndoRecoverable* implements an update–in–place policy in which an object's state is updated directly by the operations of the object class implementation, but in addition, operation records are created and logged which can "undo" the effects of these operations should it become necessary to revert to a prior state.

The class *RedoRecoverable* implements a deferred update policy in which operations on an object do not actually update the state of the object until the recovery region is successfully exited (i.e., transaction commit). Until that point, the current state of the object is determined by the initial state plus the set of "redo" operations which have been logged against it since the establishment of the recovery region. This is a conservative policy which may be applicable in restricted circumstances when "undo" is impossible e.g., the missile-firing example given in Chapter Two.

The class *CompRecoverable* implements a forward recovery policy for which compensating actions are recorded corresponding to each operation on an object. If recovery becomes necessary, the compensating actions are employed to return the object to a consistent state. The only difference between the compensating action and the *UndoRecoverable* class is in the semantics of the state of the object after recovery: the *UndoRecoverable* class reverts to a prior state - specifically, to the state which the object had at the start of the recovery region. The *CompRecoverable* class restores the object to some consistent state, although it may not be the same as the state at the start of the action - indeed, it may not be a state the object has ever been in before i.e., not_necessarily a prior state.

Note that the operation-based classes do not inherit from *CheckPointObject* and hence there is no requirement for these classes or their descendants in the class hierarchy to implement the state capture and restoration routines *SaveState* and *RestoreState*. Instead, the operation-based recovery classes impose a different obligation on the sub-class implementer: the implementation and registration of appropriate undo, redo, or compensating actions.

4.2.9 Concurrency Control

The most widely studied form of locking is conservative two-phase locking [Bernstein *et al* 87]. Many alternatives or extensions have been proposed e.g., non-strict two-phase locking, time-stamp ordering, optimistic two-phase locking. While these alternative locking strategies may be excellent for specific classes of objects, they all suffer some disadvantage in recovery (e.g., cascaded aborts), distribution (e.g., central timestamp generator) or correctness (e.g., by breaking serializability or atomicity) that limits their general applicability. Since conventional transaction processing systems are not flexible enough to vary the concurrency control method by object class or workload, they typically adopt a single general purpose policy such as two-phase locking. By encapsulating the semantics of concurrency control in an object class, *Shared*, as in Figure 4–5, it is possible, using inheritance and polymorphic operations, to gain back the flexibility to vary the concurrency control policy by object class.

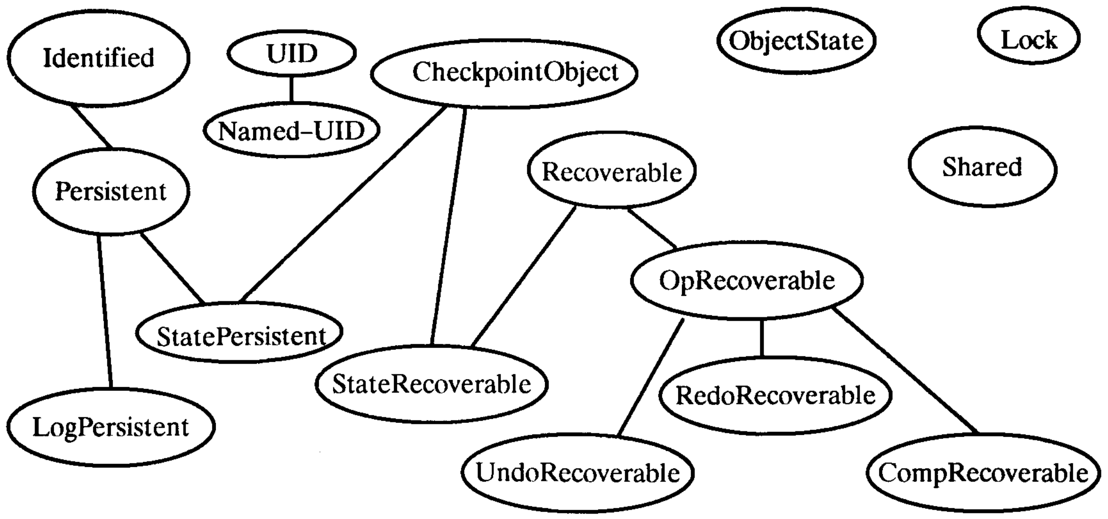


Figure 4–6: Class hierarchy including locks

To define a concurrency control class that supports two–phase locking, it is necessary to define a *Lock* class. The *Lock* class encapsulates the lock conflict information and implements a specific locking policy (e.g., simultaneous read, exclusive write). Even within the protocol of two–phase locking, there may a requirement for different lock types for different classes of object. This requirement is met by allowing new *Lock* classes to be derived from the basic *Lock* class. These derived classes may inherit from *Recoverable* or *Persistent* if these properties are desired for the *Lock* class. The concurrency controller embodied by the *Shared* class can continue to manage operation invocation conflicts by invoking polymorphic operations defined by the basic *Lock* class.

4.3 Transactions

In the spirit of the object programming model, transactions can be defined by a class of objects exporting the operations, *Begin, Commit* and *Abort*. The question is, given the hierarchy shown in Figure 4–6, where should such a class be derived? In conventional transaction systems, there is a certain amount of state associated with each transaction which must be recorded on stable storage. This data, sometimes called the *intentions list*, is usually written to a *transaction log* [Bernstein *et al* 87] which contains a (roughly) time-ordered sequence of transaction records. In the model presented here, the class *Transaction* can simply be derived directly from the class *Persistent* (see Figure 4–7), the intentions list being captured as the permanent state of the transaction record. The transaction data itself is not recoverable so there is no need to derive from the class *Recoverable*. Since the information associated with each transaction is encapsulated in a separate object, there is no need to apply concurrency control to these objects (unlike a transaction log). In effect, the concurrency control problem has been shifted to the object store, where concurrent accesses to persistent object states are serialised.

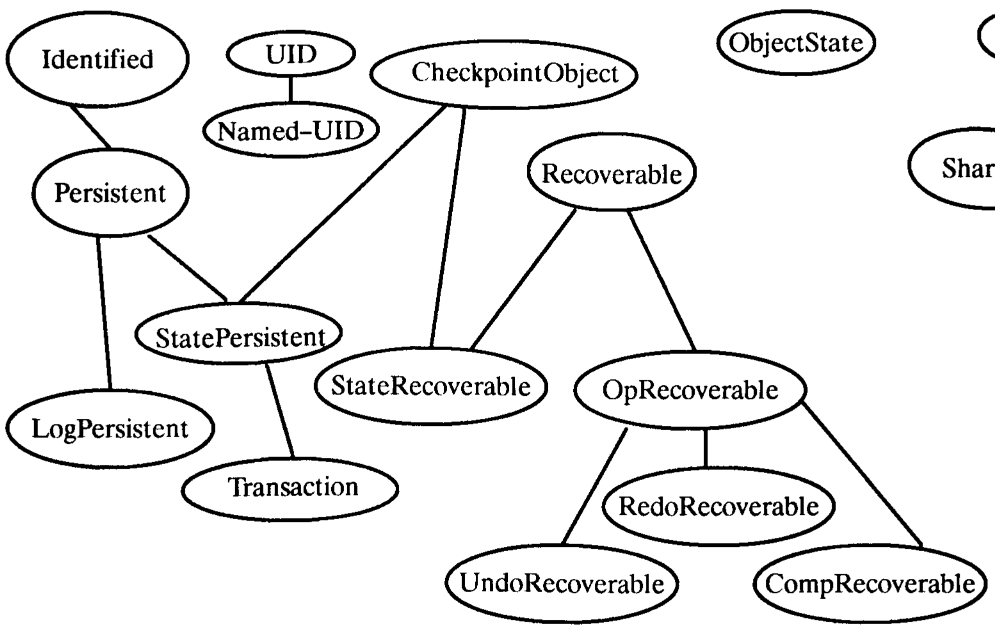


Figure 4–7: Class hierarchy including transactions

4.3.1 Integrating Transactions with other classes

The properties exported by classes *Persistent*, *Recoverable*, and their descendants in the class hierarchy are useful in their own right. As has been shown by the positioning of the class *Transaction*, a programmer might define a class of persistent objects which are not shared or recoverable. The concept of a transaction is a boundary definition, a grouping of operations. In a transaction processing environment there is a definite protocol imposed on the invocation of the operations of these property classes. The *Begin* operation of the class *Transaction*, defines the starting boundary of a recovery / atomicity / persistence region. All operations between the start and end of the transaction on *Persistent, Recoverable* or *Shared* objects are grouped together to provide the atomicity of transactions. When an (top-level) transaction commits, for example, locks should be released (an operation of the *Shared* class), permanent state should be recorded on stable storage (an operation of the *Persistent* class) and recovery state should be discarded (an operation of the *Recoverable* class). In fact, all the operations of these classes should be invoked *only* in respect of state changes to the enclosing transaction (i.e., an individual object should not release locks mid-transaction). How can we arrange for the correct operations to be invoked at the correct times?

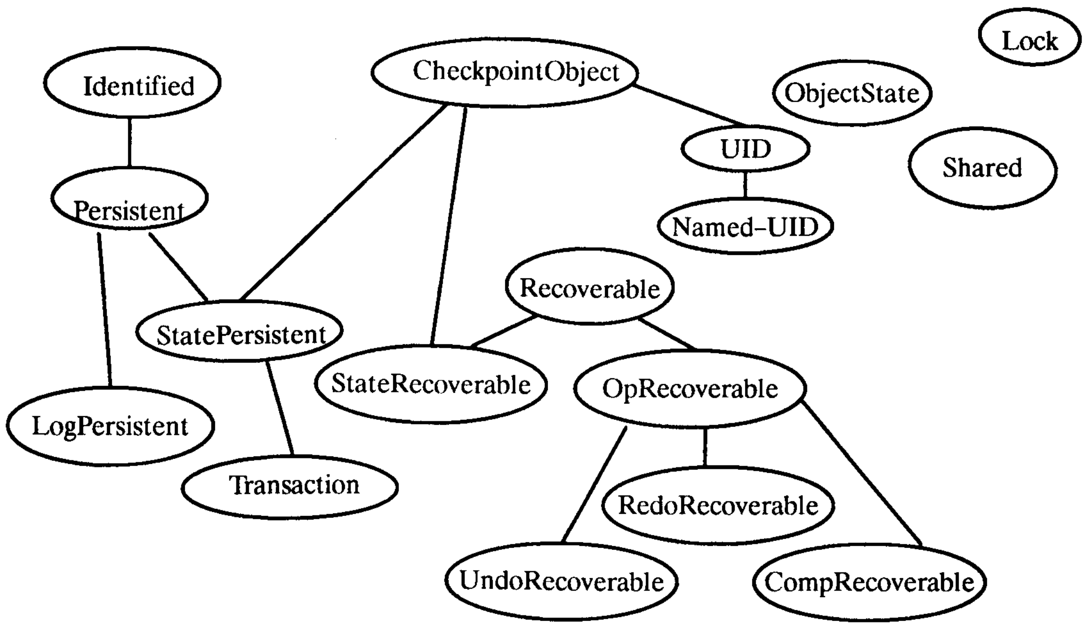


Figure 4–8: Complete class hierarchy[1]

4.3.2 Detecting Transaction State Changes

One possibility for co-ordinating the operations of the transaction with the operations of the property classes, would be to explicitly insert the necessary invocations on the property classes with the transaction invocations. For example, after invoking *Transaction.Commit*, the application could invoke each of *Shared.ReleaseAllLocks*, etc. If application programs were actually written in some higher-level language which was translated down into C+ +, this option might be acceptable. However, in the absence of such a preprocessor, it is necessary to devise some way to trigger the correct operations "automatically" for classes

1 Note that UID and NamedUID are shown in their actual position in the hierarchy as subclasses of CheckpointObject. This is because these classes are "checkpointable" and used in the implementation of state-based classes.

derived from these property classes (e.g., *StatePersistent*). In any case, the transaction object must record the identity of the objects which were affected by the action in the intentions list so that server faults during the commit process can be tolerated.

After the start of a transaction, every object upon which operations are invoked becomes part of the *action group* for that transaction. When the transaction completes, either successfully or unsuccessfully, there is a need for the object of class *Transaction* to inform the members of the action group of the result of the transaction. Since the set of objects accessed between the start and end of the transaction will be determined by the control flow of the application and cannot generally be determined statically, *Transaction* objects will maintain the action group. At the start of a transaction, the action group is empty. As operations are performed on objects, those objects must be added to the set to be informed at transaction completion. This implies that an operation of the class *Transaction* (e.g., AddToActionGroup) gets invoked each time any operation is invoked on some application object which is derived from a "property" class.

As mentioned above, each of the classes, *Persistent, Recoverable* and *Shared* and their descendants, already imposes a requirement on sub-class implementers to notify them of key events. For example, the state-based persistence class *StatePersistent* requires a sub-class implementer to indicate the operations that will modify the state of the object; the lock-based concurrency control class *Shared* requires a sub-class implementer to take an appropriate lock at the start of every operation. These super-class operations can, in addition to providing their stated behaviour, also notify any transaction in progress that the object has been accessed, is participating in the transaction, and should be notified about transaction state changes. In this way, without additional effort on the part of application class developers, objects can be dynamically added to the action group maintained by the transaction object.

Although some objects may be accessed without being added to the action group, those accesses are by definition accesses that do not affect transaction semantics. For example, for a class of objects derived from *Persistent* but not from *Shared*, read operations will not notify the *Persistent* class (it is only notified if the state is modified). However, transactions, whether committed or aborted, have no effect upon read-only access to persistent (non-shared) objects. If an object **is** shared, the sub-class implementer will have acquired a read lock for the operation and the object will be added to the action group maintained by the transaction. The lock will be released when the transaction terminates because the transaction object will invoke an operation on the application object which will in turn release locks. By this mechanism, overhead for transactions and objects is kept to a minimum according to the selected properties ascribed to an object.

4.4 Summary of Class Hierarchy Development

The class hierarchy shown in Figure 4-8 has all the necessary functionality to support transaction processing for persistent objects. An applications developer can derive new object classes with precisely the properties desired. By instantiating objects of the class *Transaction*, the applications developer can create transactions to manipulate objects of these classes.

The class hierarchy is complete for non-distributed transaction processing with persistent objects. The addition of distribution, replication and migration properties would extend the breadth of the hierarchy but would not require any structural changes.

The remainder of this chapter details the operations defined for each of the classes in the hierarchy. The semantics of these operations are carefully defined to retain independence among the classes – to maintain the *pure* semantics suggested by the class names in this informal introduction.

4.5 Class Definitions

In the descriptions that follow, the actual class definitions are presented along with explanatory text. Although the definitions are shown in C+ +, the same classes and semantics could be used in other object-oriented languages. Some of the difficulties of "mixing" properties could be alleviated by improved language support offered by some object-oriented languages, but no significant change to the hierarchy would be required or enabled by these features.

Most of the operations shown below are defined to return as their value an ErrorCode. Since the programming language in which these classes are implemented (C+ + as implemented in AT&T CFront, version 2.1) does not support exception handling, an explicit status code is returned from each operation. In an improved version of C+ + or any other language where exception handling was available, the operations would generally return no value, raising exceptions when errors were encountered that needed to be reported to the invoker.

4.5.1 Checkpoint Object

The class *CheckpointObject* defines two abstract functions *SaveState* and *RestoreState*. These operations, which are intrinsic to some object-oriented languages, provide a means for saving the state of an object in a form suitable for storage on disk or transmission across a network. That is, these functions must encode any address-space relative information (e.g., pointers) in an address-space independent fashion such as UIDs.

```
class CheckPointObject {
protected:
    virtual ErrorCode SaveState(ObjectState&) const=0;
    virtual ErrorCode RestoreState(ObjectState&)=0;
};
```

Figure 4-9: Class CheckpointObject

- **SaveState** encodes the current state of the object into an ObjectState object (essentially a buffer). This operation is sometimes called *marshalling* in RPC-related literature [Sun 88]. The state of the object is encoded to eliminate references to volatile objects e.g., memory addresses, replacing them with permanent identifiers. The encoded form of the object is not required to resemble the natural structure of the object. Note that the state of the object is *saved* into a buffer only; this operation does not imply any disk or other stable storage operation.

- **RestoreState** is the inverse operation of SaveState. Also known as *un–marshalling* in RPC–related literature, RestoreState simply decodes the information in the ObjectState and initialises the state of the object accordingly.

4.5.2 ObjectState

An *ObjectState* object is a buffer containing the "serialised" state of another object. It is the *ObjectState* which is read from and written to stable storage. The class *ObjectState* defines only two operations: *Pack* and *Unpack*. These operations are overloaded for all primitive types in the host language. For example, in C++, the Pack and Unpack operations are defined for the basic types: int, long, float, double and char and unsigned variants of these. For all objects of class *CheckpointObject*, *Pack* (object) invokes *Object.SaveState()*. Similarly, *Unpack (Object)* invokes *Object.RestoreState*. In this way, the operations *Pack* and *Unpack* are defined for all primitive types and for all classes derived from class *CheckpointObject*. *Pack* and *Unpack* are also defined for class ObjectState, as if it were a primitive type. Although it could logically be derived from *CheckpointObject* thus inheriting these operations, such a definition may run afoul of circularities in the operation definitions for the two classes.

```
enum { RecoveryState, StorageState } ObjectStateType;

class ObjectState {
public:
    ObjectState(ObjectStateType,unsigned char *,unsigned long);
   ~ObjectState();
    ObjectStateType Type();
    unsigned long Size() const;
    const unsigned char *Buffer() const;

    ErrorCode Pack(const ObjectState &);
    ErrorCode Pack(const CheckPointObject &);
    ErrorCode Pack(char);
    ErrorCode Pack(unsigned char);
    ErrorCode Pack(int);
    ErrorCode Pack(unsigned int);
    ErrorCode Pack(short int);
    ErrorCode Pack(unsigned short int);
    ErrorCode Pack(long int);
    ErrorCode Pack(unsigned long int);
    ErrorCode Pack(float);
    ErrorCode Pack(double);

    ErrorCode Unpack(ObjectState &);
    ErrorCode Unpack(CheckPointObject &);
    ErrorCode Unpack(char &);
    ErrorCode Unpack(unsigned char &);
    ErrorCode Unpack(int &);
    ErrorCode Unpack(unsigned int &);
    ErrorCode Unpack(short int &);
    ErrorCode Unpack(unsigned short int &);
    ErrorCode Unpack(long int &);
    ErrorCode Unpack(unsigned long int &);
    ErrorCode Unpack(float &);
    ErrorCode Unpack(double &);
private:
    ObjectStateType MyType;
    unsigned char *buffer;
    unsigned long size, used, start;
    ErrorCode PackBytes(unsigned char *, unsigned long);
    ErrorCode UnpackBytes(unsigned char *, unsigned long);
};
```

Figure 4–10: Class ObjectState

- **Pack** is defined for all primitive (scalar, non–pointer) types in C++. It encodes the value of its parameter in a representation which is "suitably independent" of machine architectures and implementations. For example, an implementation might encode integers in "network byte order" to overcome differences in byte–ordering, but employing a standard word size (e.g., 32–bits). Another implementation of Pack might cater to different word sizes as well by storing additional information. In its full generality, Pack might employ some international standard encoding scheme such as ASN.1 [Steedman 90].

As described above, when Pack is invoked with an object of a class derived from CheckpointObject, it invokes the SaveState operation of that object to Pack the state.

- **Unpack** is the inverse operation of Pack. The ObjectState instance on which it is invoked interprets the next bits of the saved state according to the type of the (output) parameter. Type checking information may be encoded (by the Pack operation) and decoded here to detect incorrect usage.

  Unpacking a CheckpointObject is accomplished by invoking the object's RestoreState operation.

4.5.3 Unique Identifier

A Unique Identifier is an object which can be distinguished from all other objects of its type created anywhere in the network at any time. It typically consists of a number of 64–128 bits which is composed by appending a local sequence number or time–stamp to a some kind of (compact) host identifier (e.g., a machine serial number). The resulting number is unique to the local machine (because of the sequencing) and unique in the network because of the host identifier. As long as the sequence numbers are monotonically increasing, there will never be two Unique Identifiers (UIDs) with the same number. It is desirable that the UID can be locally generated without consulting other nodes since UIDs are used often and need to be generated efficiently. The scheme described here has this property. Also, it is desirable that no interpretation be applied to the contents of the UID. This allows for other generation techniques (e.g., random numbers) and facilitates migration of objects.

```
class UID : public virtual CheckPointObject {
public:
    UID();                      // Generate a new unique identifier
    UID(char *);                // Parse a string containing a UID
    virtual unsigned long Hash() const;
    virtual operator==(const UID &u) const;
    virtual operator!=(const UID &u) const;

    virtual istream & Parse(istream &);
    virtual ostream & Print (ostream &) const;
protected:
    virtual ErrorCode SaveState(ObjectState &) const;
    virtual ErrorCode RestoreState(ObjectState &);
private:
    unsigned long value[UID_LONG_COUNT];
};

extern const UID UID_NIL;
```

Figure 4–11: Class UID

The two forms of constructor for UID are used to handle new and existing UIDs.

- **Hash** returns a hash value for a UID which is used in various tables as a key. Although this function could be provided separately, it is so frequently used and useful with UIDs that it was placed as a class member.
- The comparison operators, **operator= =** and **operator!=** are defined to perform simple bit–wise comparison of UID values. These are the only operators defined for UIDs, hence, for example, sorting UIDs is not possible since no ordering operations are provided.
- **Print** and **Parse** are defined to allow easy transition of UIDs to/from a printable form.

Since class *UID* is derived from *CheckpointObject*, it must implement the virtual operations, *SaveState* and *RestoreState*. This is simply a convenience for the users of the class *UID* so that they can *Pack* UIDs directly into *ObjectStates* rather than having first to convert them to strings.

4.5.4 NamedUID

UIDs are system generated to ensure uniqueness. These UIDs are typically difficult for people to remember and awkward to type because they are such large numbers. People would rather assign names to objects which could be mapped to UIDs. The class NamedUID defines just such objects. NamedUIDs are objects that contain a string name which is mapped to a UID using a *name server.* The structure of names (e.g., context identifiers, naming hierarchies, naming syntax) is independent of this class specification.

```
class NamedUID : public UID {
public:
    // Create a new mapping from string to UID
    NamedUID(const char *, const UID &);

    // Lookup an old mapping (UID_NIL if not found)
    NamedUID(const char *);
};
```

Figure 4–12: Class NamedUID

4.5.5 Identified

The class *Identified* provides identity for objects that is independent of the contents (i.e., *value*) of the object and independent of the address-space (i.e., *context*). A member of this class (or a derivative class) has an identity which is unique from all other objects on all other machines everywhere, and permanent (i.e., the *identity* is permanently assigned to the given object). This is the class of objects that *contains* a (single, identifying) UID. Note that this is different from the class UID which defines the actual identifier objects. The new operations at this node of the class hierarchy are:

```
class Identified {
public:
    const UID &GetUID() const;
    const UID &GetTypeUID() const;
protected:
    Identified(const UID &typ);
    Identified(const UID &typ, const UID &u);
private:
    UID uid;
    UID typeuid;
};
```

Figure 4–13: Class Identified

The default constructor for the Identified class creates a new object and assigns a new unique identifier to it.

The alternate form of the constructor, taking a UID as a parameter, creates an object and assigns the specified unique identifier to it. Note that this operation simply gives the specified identity to the object; no other change is made to the state of the object.

- **GetUID** returns the object's unique identifier (which was assigned when the object was first constructed).

4.5.6 Recoverable

The class *Recoverable* defines operations required for the management of recovery regions:

```
class Recoverable {
public:
    Recoverable();
    ~Recoverable();
    virtual ErrorCode Establish();
    virtual ErrorCode Recover();
    virtual ErrorCode Discard();
protected:
    int RegionIndex;
};
```

Figure 4–14: Class Recoverable

- **Establish** creates a new recovery region for the object.
- **Recover** ends a recovery region and *restores* the object state (in the case of backward recovery) or at least leaves it in some consistent state.
- **Discard** ends a recovery region leaving the object state alone (as of the last operation that modified it) and discarding any recovery state information that was associated with the recovery region.

These three operations are normally not invoked by an application program that is using transactions. The *Recoverable* object will be enrolled as an active object in the transaction's action group as necessary. The *Establish* operation will be called as necessary (logically at the start of the transaction in which the object is modified). When the transaction completes, either *Recover* or *Discard* will be invoked depending on whether the transaction aborted or committed.

4.5.7 StateRecoverable

StateRecoverable is a kind of *Recoverable* which relies on "before image" state capture to implement recovery.

```
class  StateRecoverable : public virtual Recoverable,
                          public virtual CheckPointObject {
public:
    StateRecoverable();
    virtual ~StateRecoverable();

    virtual ErrorCode Discard();
    virtual ErrorCode Recover();
protected:
    class StateRecoveryMgr;
    StateRecoveryMgr *mgr;

    virtual ErrorCode Modified();
};
```

Figure 4–15: Class StateRecoverable

The operation *Establish* from class *Recoverable* is not redefined at this level. The parent class behaviour is sufficient. The operations *Recover* and *Discard* however need to be redefined in *StateRecoverable* to implement the correct behaviour.

- The protected operation **Modified** must be called by the sub–class implementer prior to the first modification of the object state in any operation. The *Modified* operation causes the object to be added to the action group for any transaction in progress and causes a snapshot of the "before image" of the object to be taken (if necessary) using the *CheckpointObject* operation *SaveState.*
- **Recover** pops the top checkpoint from the recovery stack and *restores* the object state from that checkpoint using the *CheckpointObject* operation *RestoreState*.
- **Discard** pops the top checkpoint from the recovery stack and discards it.
- **Establish** is inherited from the class *Recoverable* and is not redefined at this level of the class hierarchy

As described above for the class *Recoverable*, the operations *Recover* and *Discard* will not normally be invoked directly by applications programs. They will invoked as a result of transaction state changes. Only the operation *Modified* will be directly invoked and then only by a sub–class implementer creating a *StateRecoverable* class.

4.5.8 OpRecoverable

OpRecoverable is a kind of *Recoverable* which relies on operation logging to implement recovery. Since all of the different kinds of operation based recovery require sub–class developers to define operations which can be invoked (at times appropriate to the recovery technique), an abstract class *Operation* is defined which has a single (pure virtual) operation *Perform*. Sub–class developers implementing application classes using operation–based recovery must derive suitable sub–classes of *Operation* to capture enough information to perform the recovery operation. For example, if undo–based recovery is chosen, a sub–class developer will derive the application class from *UndoRecoverable* and also derive application–specific *Operation* sub–classes for use in recovery. The *Operation* sub–classes need not be visible to any user of the application class – they form part of the implementation.

```
class Operation {
public:
    virtual ErrorCode Perform()=0;
};


class OpRecoverable : public virtual Recoverable {
public:
    OpRecoverable();
    virtual ~OpRecoverable();
protected:
    class OpRecoveryMgr *mgr;
    virtual OpRecoveryMgr *MakeMgr(int idx)=0;

    virtual ErrorCode AddOperation(Operation *);
};
```

Figure 4–16: Class OpRecoverable

The operations *Establish, Recover* and *Discard* from class *Recoverable* are not redefined at this level. They may be redefined by sub–classes of *OpRecoverable* that embody specific recovery techniques.

- The protected operation **MakeMgr** is used only by the recovery classes *UndoRecoverable, RedoRecoverable,* and *CompRecoverable* and is provided only as an implementation convenience.
- **AddOperation** is the operation which sub-class developers invoke to record a new recovery operation. The class *OpRecoverable* maintains this sequence of recovery operations for use by the specific recovery sub-classes, *UndoRecoverable, RedoRecoverable* and *CompRecoverable.*

4.5.9 UndoRecoverable

UndoRecoverable is a kind of *OpRecoverable* which allows operations to update the object state in-place, recording a corresponding undo operation in each case. If recovery is necessary, the undo operations are *Perform*ed to restore the object to its prior state.

```
class  UndoRecoverable : public virtual OpRecoverable {
public:
    UndoRecoverable();
    virtual ~UndoRecoverable();

    virtual ErrorCode Discard();
    virtual ErrorCode Recover();
protected:
    virtual OpRecoveryMgr *MakeMgr(int idx);
};
```

Figure 4–17: Class UndoRecoverable

The operation *Establish* from class *Recoverable* is not redefined at this level. The parent class behaviour is sufficient. The operations *Recover* and *Discard* however need to be redefined in *UndoRecoverable* to implement the correct behaviour.

- Each operation of a class derived from *UndoRecoverable* must register "undo" operations.
- **Recover** performs the undo operations, starting with the most recent "done" operation and proceeding to the least recent (i.e., LIFO or stack order).
- **Discard** simply discards all the undo operations as the state of the object is already correct for the *committed* case.

As described above for *Recoverable*, *Recover* and *Discard* will not normally be invoked directly by applications programs. They will be invoked as a result of transaction state changes. Only the operation *AddOperation* (inherited from the class *OpRecoverable)* will be directly invoked to register undo operations and then only by a sub–class implementer creating a new class which is derived from the *UndoRecoverable* class.

4.5.10 RedoRecoverable and CompRecoverable

The classes *RedoRecoverable* and *CompRecoverable* are very similar to *UndoRecoverable* in their interfaces and implementations, differing only in the implementation of operations *Recover* and *Discard*. *RedoRecoverable* implements a deferred update policy such that state changes are only applied if and when the transaction commits. Thus, *RedoRecoverable.Discard* actually applies the operations to the state at commit time; *RedoRecoverable.Recover* simply discards the operations. The class *CompRecoverable* has the same implementation as *UndoRecoverable*: *Discard* discards the operation list and *Recover* applies it. The difference between *UndoRecoverable* and *CompRecoverable* is in the semantics of the class – providing backward or forward error recovery respectively.

4.5.11 Persistent

The class *Persistent* defines operations required to manage the persistent state of the object in the context of a transaction system.

```
class Persistent : public virtual Identified {
public:
    Persistent(const UID &typ, const UID &);
    Persistent(const UID &typ, const char *hostname=0);
    ~Persistent();
    virtual ErrorCode Modified();
    virtual ErrorCode Initialize();
    virtual ErrorCode UnCatalog();
    virtual ErrorCode ForceWrite();
    virtual ErrorCode Terminate();
protected:
    virtual ErrorCode SaveState(ObjectState&) const;
    virtual ErrorCode RestoreState(ObjectState&);
private:
    long version;
    bool modified;
    ObjectStore MyObjectStore;
    ErrorCode OnPrepare(const UID &);
    ErrorCode OnCommit();
    ErrorCode OnAbort();
    PersistenceActionManager *mgr;
};
```

Figure 4–18: Class Persistent

There are two forms of constructor for persistent objects: one takes a UID and accesses an existing persistent object; the other takes a host name an creates a new persistent object at that host location. The new instance of the class will actually become permanent if and when the enclosing (top–level) transaction commits. If the enclosing action does not successfully commit, the object created by this constructor will not be made permanent.

- The **UnCatalog** operation removes the object from the persistent store and prevents its state from being recorded at transaction commit time.
- The **OnPrepare** operation ensures that the current state of the object is recorded on stable storage. For implementations that perform updates in place and log operations, there is little work to do to prepare. For implementations that do not update in place, the Prepare operation must write the current state of the object into the permanent object store provisionally. This ensures that the current state of the object (in volatile memory) is safely stored in stable storage. However, subsequent attempts to

read the state will return the old state until and unless a Commit operation is applied.

- The **OnCommit** operation makes a provisional update of the permanent store take effect permanently. The old state of the object in the object store, if any, is discarded and replaced by the provisional state. Subsequent attempts to read the state of the object will return the new state. In an implementation that uses update in place and operation logging, the effect of the commit operation is only to make the updated state of the object visible.
- The **OnAbort** operation discards the provisional update, if any, which was made to an object by the Prepare operation. Subsequent attempts to read the object will return the last committed state. In an operation logging situation, this operation will trigger undo operations to restore the previous state of the object.

The first three of these operations, the two forms of constructor and the UnCatalog operation are the only *Persistent* operations that an applications program will normally invoke (directly). The constructors provide the means for instantiating persistent objects and the UnCatalog operation provides the means for deleting an object from the permanent object store. The last three of these operations, Prepare, Commit, and Abort, are normally invoked as part of a transaction commit or abort operation.

4.5.12 Lock

The class *Lock* is defined independently from the class *Shared* to allow applications programmers to derive application-specific lock types. The class *Lock* provides shared-read, exclusive-write locks in its default implementation. Naturally, to support type-specific locking requirements, different lock types can be derived from the *Lock* class using the inheritance mechanism. Although the basic concurrency control model is based on a strict two-phase commit protocol, any type of lock can be accommodated. The operations supported by the *Lock* class are as follows:

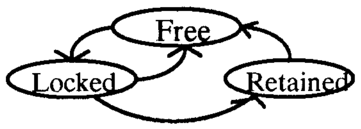
```
enum { ReadLock, WriteLock } LockType;
enum { FreeLock, SetLock, RetLock, UnInitLock } LockStatus;

class Lock : public CheckPointObject {
public:
    Lock();
    Lock(LockType t,const UID &own);
    virtual bool Conflicts(const Lock&) const;
    virtual LockType GetType(void) const;
    virtual const UID &GetOwnerID(void) const;
    virtual void SetOwnerID(const UID &);
    virtual LockStatus GetStatus(void);
    virtual void SetStatus(LockStatus);
    virtual operator==(const Lock &l) const;
    virtual operator!=(const Lock &l) const;
protected:
    virtual ErrorCode SaveState(ObjectState&) const;
    virtual ErrorCode RestoreState(ObjectState&);
private:
    LockType   Type;
    LockStatus Status;
    UID        OwnerID;
};
```

Figure 4–19: Class Lock

The constructor for a Lock takes, as input parameters, the UID of the owner of the lock, and the type of the lock (e.g., read or write). In the context of transactions, the UID identifies the transaction on which the lock depends. In another context, this might be the UID of a thread of control (e.g., a simple multi-tasking application without transactions).

- **Conflicts** returns true if the lock object conflicts with the other lock object given as an input parameter.
- **GetOwner** returns the UID of the owner of the lock.
- **SetOwner** changes the ownership of the lock. This operation is used to manage the locks held by nested transactions at the time that they commit.
- **GetStatus** returns the current status of the lock.

- **SetStatus** changes the state of the lock according to the following state transition diagram:



For a further explanation of these lock modes, consult [Moss 81].

Figure 4–20: Lock modes and permissible transitions for nested transactions

4.5.13 Shared

It is now possible to explain the operations provided by the class *Shared*. This class provides the locking operations, SetLock, ReleaseLock, ReleaseAll and PropagateLocks.

```
class Shared {
public:
    Shared();
    virtual ErrorCode TakeLock(Lock *l);
    virtual ErrorCode ReleaseLock(Lock&);
    virtual ErrorCode ReleaseAll (const UID &);
    virtual ErrorCode Propagate (const UID &, const UID &);
private:
    LockList locks;
};
```

Figure 4–21: Class Shared

- **SetLock** attempts to apply a lock to an object by comparing the desired lock with other locks already held on the object. This comparison is performed by repeated invocations of the *Conflicts* operation provided by the lock class. The *Conflicts* operation checks for conflicting *lock modes*, the SetLock operation may take lock *owners* into account in determining whether or not conflicting modes are acceptable. For example, in [Moss 81], nested transactions may take locks previously held in a '*conflicting*' mode by another nested transaction within the same top-level action.
- **ReleaseLock** changes the lock status to 'FREE' and removes the lock from the list of locks held on this object.
- **ReleaseAll** releases all locks associated with a particular owner.
- **Propagate** changes the ownership of all locks with owner, UID_1, to be owned by UID_2. This operation is normally invoked by the transaction system at the commit phase of a nested transaction to propagate the locks to the parent transaction.

4.5.14 Transaction

Transaction objects may be declared in any declaration context in the programming language just like other kinds of objects. To increase programming flexibility, the transaction is initially 'inactive'. The transaction boundaries must be explicitly established by invoking the Begin operation of the object. Hence, merely declaring a transaction object in some scope does **not** begin a transaction for that scope. Similarly, transactions must be explicitly terminated by invocation of either the commit or abort operation. It is an error to destroy a transaction object while the transaction which it represents is still active.

```
enum {  DefinedAction, ActiveAction, InTransitionAction,
        CommittedAction, AbortedAction } ActionStatus;

enum { IndependentAction, ContextAction } ActionType;

class Transaction : public Persistent {
public:
    static Transaction *Current();  // Current action of this process
    Transaction(ActionType t=ContextAction);
    Transaction(UID &u);
    ~Transaction();
    Transaction *GetParent()  const;
    ActionStatus GetStatus() const;
    ActionType GetType() const;
    virtual ErrorCode Begin();
    virtual ErrorCode Commit();
    virtual ErrorCode Abort();
    virtual ErrorCode Join(const ActionMgr &);
protected:
    virtual ErrorCode SaveState(ObjectState &) const;
    virtual ErrorCode RestoreState(ObjectState &);
private:
    ActionType Type;
    ActionStatus Status;
    Transaction *Parent;
    ActionManagerSet Managers;
    ErrorCode Propagate();
    bool IsNested() const;
};
```

Figure 4–22: Class Transaction

- **Begin** starts the transaction. Until a Commit or Abort is executed, the transaction state will be 'active'.

- **Commit** terminates the action, invoking all the action managers in the set to notify them of successful completion of the action.
- **Abort** terminates the action, invoking all the action managers in the set to notify them of unsuccessful completion of the action.
- **Join** adds an object to the action group for the transaction. This operation is invoked "automatically" by the property classes on behalf of user objects.

4.6 Class Hierarchy Summary

This chapter has explained the development of a class hierarchy which provides applications programmers with the ability to selectively apply the distribution and persistence transparencies to application object classes. The next chapter outlines several different example applications, illustrating the way in which this programming interface can be used to develop reliable distributed applications by selective application of distribution and persistence transparencies.

5 Developing Reliable Distributed Applications

"Our first duty is to understand the problem. Having understood the problem as a whole, we go into detail. We consider its principal parts, the unknown, the data, the condition, each by itself."
G. Polya [Polya 1957]

5.1 Introduction

How does the collection of classes defined in the previous chapter support the ultimate goal of assisting in the development of reliable distributed applications? These classes can be selectively applied to application object classes, through inheritance, to provide the key transparencies of distribution and persistence described in Chapter Two. This chapter describes the application of the ideas presented in the previous chapters from two points of view:

- design and implementation of the Transaction Services i.e., the class hierarchy described in the previous chapter
- use of the class hierarchy for the development of reliable distributed applications

Section 5.2 discusses the issues involved in the design and implementation of the "property" classes. Section 5.3 briefly describes an implementation of the class hierarchy and supporting systems which has been developed to validate these ideas. Section 5.4 details the issues involved in the use of such a system for applications development. Taking the view of a class designer, several small example classes are defined showing the application of selective transparency through multiple inheritance. In each case, the class definition is given with an explanation of the properties being inherited. Some of the more interesting fragments of the implementation are presented. For clarity and conciseness, obvious or repetitive parts of the implementation are omitted.

5.2 Design and Implementation Issues

There are many small decisions to be made in the design of transaction services, however, the four discussed here have such a large impact that they have been given special attention. These four key design issues are:

- Heterogeneity – how does heterogeneity affect the design of the classes and support services described in the previous chapters?
- Object lifetimes – how to resolve the conflict between lexically scoped languages like C+ + and dynamically scoped transactions?
- Commit protocols and nested transactions– what are the interactions of commit protocols with mechanisms for recovery, persistence and concurrency control?
- Buffer management and recovery techniques – what are the combinations of persistence, recovery and concurrency control mechanisms that make sense?

To summarise some of the other design issues previously discussed for which decisions have been taken: We are considering a programming environment based on an object-oriented programming language that uses synchronous operation invocation. The transaction services interface provides a means for defining classes of objects that have some subset of the properties of persistence, recoverability, and concurrency control. Concurrency control is provided by locking using a strict two-phase locking policy although other concurrency control techniques could be employed. Persistence is provided by simple checkpointing or by checkpointing augmented with logging. Recovery is provided by shadowing (state-based recovery) or some form of operation logging. The transaction services interface also provides a means for expressing transaction start and end boundaries, permitting nested and concurrent sub-transactions. Transactions employ a two-phase commit protocol and support both upward and downward inheritance of locks. Remote object invocations will employ a Remote Procedure Call protocol, expecting at most one response for each request, after network level error handling. Communications and system faults and explicit transaction aborts will be handled by the transaction recovery mechanism. Media faults will be handled in some other way such as disk mirroring or replication. An execution environment like the one described in Chapter Three will provide underlying services of naming and locating objects, communications and stable storage.

The issue of local vs. global or remote addressing is one aspect of a general issue relating to access across machine boundaries. In homogeneous systems, that is, machines with a common hardware architecture *and* a common operating system interface, it might be possible to use some combination of machine addresses and machine identifiers to address objects anywhere in a network. In heterogeneous systems, even this is not generally possible since machine addresses will not generally have the same interpretation on different machine configurations. This is but one of the problems of heterogeneous machine support which is discussed in the following section.

5.2.1 Heterogeneity

Heterogeneity in a distributed object system raises several interesting engineering design problems. Apart from the requirement of all distributed systems that communications between machines must be possible, the issues of common data representation and, if migration is to be supported, code interpretation, must be resolved. Specifically, if objects can move among the machines of a distributed system (migration transparency), how can the representation of the data of the passive object and the operations (code) move so as to preserve the semantics of the objects? There are three aspects to this problem:

- There must be a method for capturing the state of an object in a context-independent and architecture-independent representation. The symmetric counterpart to this "capturing" operation, restoring the state of an object from some passive form, must also exist.
- Executable versions of the methods of an object must be available on all the machines of interest. This implies both a means of producing executable methods and a means of distributing them to the machines where they are needed.

- Machine-architecture independent representations of data must be obtained for transmission via messages. Architecture independence is somewhat more complex than context independence as it requires translation to and from some canonical passive representation for primitive data and a canonical representation for object composition. Higher-level protocols in the ISO OSI protocol stack, such as ASN.1, have addressed substantial parts of this data representation problem. Existing RPC systems e.g., [Bershad *et al* 87] employing such protocols provide adequate support for this mapping.

Of the three issues, the first and third seem to be adequately addressed by existing mechanisms although the difficulty of representing the type of an object across machine boundaries remains. This difficulty is related to the second issue – distributing some executable form of object methods to machines with varying hardware or software architectures. This problem has been superficially addressed by systems such as Distributed Smalltalk [Bennett 90], in which each machine has a Smalltalk interpreter and method code is sent with the objects. SOS [Shapiro *et al* 89] employs a very general mechanism of execution pre-requisites (including pre-requisite code objects) to overcome this problem. The problem is not addressed further in this thesis – it is assumed that proper versions of the executable code are available at each machine.

With a means for representing the state of objects so that all machine types in the distributed system can interpret them, and a means for distributing suitable executable code to machines that need it, the problem of heterogeneous machines can be adequately solved. Ongoing research into sophisticated interpreters and general pre-requisite facilities may lead to improved solutions for automatic distribution of executable code. Improving standards in architecture-independent data representation will also help to simplify the management of data in a heterogeneous network. This first of the four problems mentioned above seems well in hand. Turning attention to the second issue, the conflict of lexical and dynamic scoping, the solution seems less clear.

5.2.2 Object Lifetimes

There is a conflict that arises when programming language concepts of object lifetime, possibly extended to include persistent objects, interact with transaction mechanisms. The problem derives from the fact that most object-oriented programming languages are lexically scoped while transactions are inherently dynamic. To be precise, it is not scope (a visibility constraint), but extent (a lifetime constraint) of the objects which is the source of the difficulty. However, since objects are generally unreachable once program execution leaves the scope of their class definition, programming language implementations ensure that objects which are no longer reachable are destroyed either immediately, by compiler–placed code e.g., for stack allocated objects, or eventually by garbage collection. Transactions by their nature are dynamic, typically spanning multiple operation invocations on many objects. There is critical state information relating to recovery, persistence and concurrency–control which must be preserved until the transaction completes even if the lifetime of some objects involved in the transaction has expired with respect to the programming language semantics. This section describes the interactions of programming models of object lifetime and transaction requirements and the effect of these interactions on transaction system structure.

One of the distinguishing characteristics of object–oriented programming languages is the treatment of objects as autonomous entities "responsible" for maintaining their own state. Each object must be properly initialised on creation by the execution of some class–specific initialisation code, often expressed as an operation called a *constructor*. Similarly, when an object is to be deleted because its lifetime has ended, some finalisation code may be executed to tidy up referenced data structures, etc. This finalisation code may be separated into an operation called a *destructor*.

Object–oriented programming languages differ in the level of autonomous activity that they assign to objects. All objects are active for at least the duration of invocations on them. Some objects may have activity which is not related to specific client invocations. For example, in Emerald, an object operation invocation is viewed as a temporary transfer of control to an object for the execution of the invocation. However, in Emerald, each object may also contain an independent thread of control which is not related to any specific invocation. Such background activity may be, for example, to maintain internal data structures for improved efficiency in space or time.

When an object declaration comes into scope, for example by invoking an operation *f* that creates an object *o* with scope and extent local to operation *f*, storage is allocated for the object *o* and the constructor for object *o* is automatically invoked to initialise it. When the object lifetime expires, that is, when operation *f* completes, the destructor for object *o* is automatically invoked to finalise it before the storage is returned.

For persistent objects, these initialisation and finalisation operations provide a convenient place to fetch the saved state of an object and write back the state to stable storage. Just after the object's memory storage has been allocated, but before any operations are invoked on the object, the object can be initialised by loading the saved state. Finally, after all requisite operations have been performed on the object, but before the memory image of the object's state is discarded, the persistence mechanism can be invoked to capture the state of the object and record it on stable storage. Similarly, for recovery state and concurrency–control state e.g., locks, these initialisation and finalisation points are critical stages in the object lifetime where some work must be performed to provide the desired semantics.

However, when transactions are employed to provide failure atomicity and concurrency–control for objects, it is the lifetime of the transaction rather than the lifetime of the object that matters. Objects involved in an ongoing transaction must not have their state indiscriminately written to stable storage before the transaction commits, even if the executing program has no further use for the objects. A subsequent failure and consequent transaction abort might require the old state of the object to be restored or retained in the stable store.

For transient objects whose lifetime is totally contained within transaction boundaries, concurrency–control information and recovery state could be simply discarded when an object lifetime expired. What use is recovery state for an object that no longer exists? What concurrency violation could occur after an object has been destroyed? However, the possibility that an object is persistent implies that the object may not really have disappeared – just moved to a stable memory. Although a currently executing transaction (presently) has no further use for the object, and might discard the recovery state, the object could *reappear* in this same transaction later in the execution. If the object should come back into scope (extent) during the execution of the transaction, the old recovery state and concurrency–control information would be required. Indeed, until the transaction completes, the concurrency–control information is required to prevent inconsistent access by other transactions whether or not the initial transaction ever accesses the object again. Hence object state, including recovery state and concurrency control information, might need to be retained even after an object's lifetime had expired.

One way to address the problem of ensuring that objects are no destroyed until the transactions in which they are involved have ended would be to restrict the choice of possible programming languages to those that allocate all objects "in the heap" such as CLU [Liskov *et al* 79]. In such languages, the lifetime of objects is completely divorced from their scope. A transaction would presumably have some reference to the object in its action group even after the application program had ceased to refer to the object and the garbage collector would retain the object in the heap until all references were gone. Thus, the problem of early death of objects could be reduced to a problem of reliable distributed garbage collection. This is a promising future direction, although much work remains to be done before this "reduced" problem can be considered solved for general networks of machines. The problem of objects leaving extent before the end of their transactions is also complicated by the provision of nested transactions although the same basic mechanisms can be applied. In the following section, the interactions of commit protocols with the operations of the property classes are discussed.

5.2.3 Commit Protocols and Nested Transactions

Nested transactions, as described by Moss [Moss 81], permit the construction of a hierarchy of atomic actions in which sub–transactions may be aborted without aborting the enclosing transactions. An excellent database–oriented summary of the interactions of nested transactions with recovery and concurrency control including an exhaustive consideration of various design parameters is available in two papers by Haerder and Rothermel [Haerder and Rothermel 87a] [Haerder and Rothermel 87b]. However, there are two important aspects which need further consideration:

- the interaction of nested transactions with objects that have reached the end of their extent in the programming language sense

- the clear separation of persistence, recovery and concurrency control as orthogonal properties

The first issue requires object destruction to be linked with transaction state transitions. The second issue, raised in Chapter Two, is addressed here by separating the actions required for the various transparencies at transaction state transitions. Since these issues must be addressed in relation to transitions in the state of a transaction, the next section describes the nested transaction model and the state transitions of nested and top–level actions with respect to commit protocols.

As a program executes, invoking transactions, which in turn invoke sub–transactions, a hierarchy of transactions is dynamically constructed. The root of the hierarchy of transactions is referred to as a top–level transaction. The interior nodes and the leaves of the hierarchy are called sub–transactions. Nested transactions offer several advantages over flat transaction models including the possibility of intra–transaction parallelism and recovery control. Perhaps the most important advantage of the nested transaction structure however is that it matches well with current programming language technology for structuring software. Programming languages introduce dynamic modularity by the use of subroutines. This general subroutine invocation model is carried through most modern programming languages with minor variations called functions, procedures, routines, subroutines, subprograms, operations, methods, or handlers. The variations have mostly to do with parameter and result passing conventions, different scoping rules, and occasionally, asynchronous invocation. However, all of these programming language notions produce a dynamic tree or hierarchy of executions that is very similar to the transaction tree resulting from successive or concurrent invocations of nested transactions. Consider the problem of writing a reliable utility function in a flat transaction model. If the utility function employs transactions to achieve its reliability, then no caller of that utility can use transactions for such use would constitute nesting transactions when the utility function was invoked. The significance of nested transactions for integrating programming language structures with database concepts of transactions cannot be overstated.

To analyse the interactions of nested transaction semantics with the semantics of objects possessing one or more of the properties of persistence, recoverability or concurrency control, it is necessary to consider the state transitions of transactions. These state transitions are a function of the commit protocol employed. Figure 5–1 depicts the state transitions of transactions for both two–phase and three–phase commit protocols. In a "centralised commit site" variation of either two–phase or three–phase commit protocols, the actual operations that take individual transaction participants from state to state differ depending on whether one considers the transaction co–ordinator or slave. Decentralised commit site protocols have uniform actions at all sites although they incur extra message traffic to reach consensus. Nested transactions have all the same states and transitions as top–level transactions although the "final" commit of a nested transaction may be undone by an abort of enclosing action.

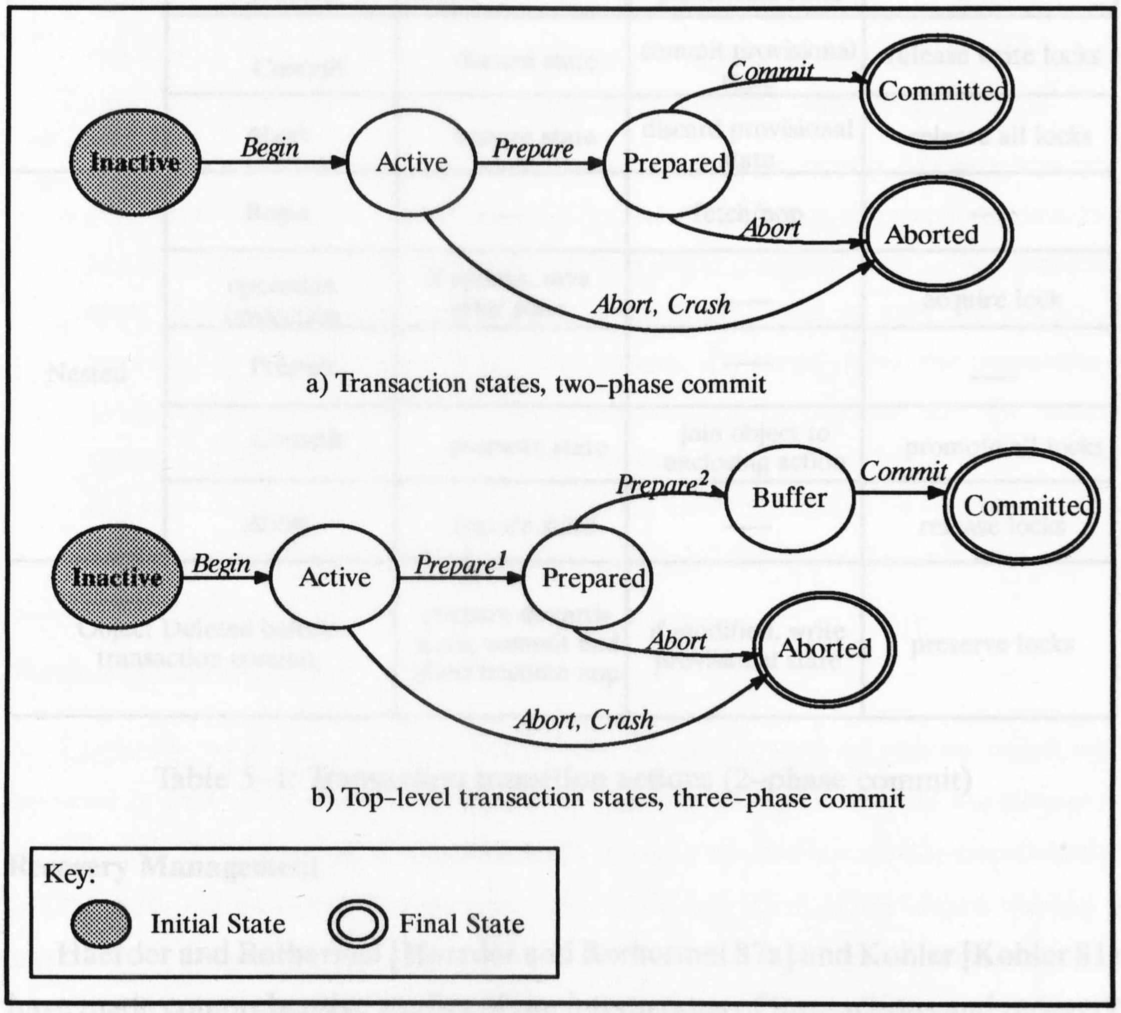


Figure 5–1: Transaction State Transitions

For each of the properties of recovery, persistence and concurrency control, the chart below (table 5–1) summarizes the action that must be taken on a transaction state change assuming that recovery is implemented using shadowing and concurrency control is achieved by locking. A similar chart can be constructed for other approaches to concurrency control, recovery, or transaction commit protocol without substantially altering the discussion. The following sections explain these actions in more detail.

		Recovery	Persistence	Concurrency Control
Top-level	Begin	----	fetch state	----
	operation invocation	save prior state as necessary	if update, mark as modified	acquire lock
	Prepare	----	if modified, write provisional state	release read locks
	Commit	discard state	commit provisional state	release write locks
	Abort	restore state	discard provisional state	release all locks
Nested	Begin	----	fetch/nop	----
	operation invocation	if update, save prior state	----	acquire lock
	Prepare	----	----	----
	Commit	promote state	join object to enclosing action	promote all locks
	Abort	restore state	----	release locks
Object Deleted before transaction commit		prepare discards state, commit and abort become nop	if modified, write provisional state	preserve locks

Table 5–1: Transaction transition actions (2–phase commit)

Recovery Management

Haerder and Rothermel [Haerder and Rothermel 87a] and Kohler [Kohler 81] have made comprehensive studies of the interactions of transactions and recovery techniques in database systems – the former deals especially with variations of design parameters for nested transactions and the latter focuses on distributed systems. The recovery actions shown Table 5–1 can be explained as follows:

- Logically, the state of an object involved in a transaction is "captured" when the transaction begins. Actually, the capture operation can be deferred until just before the first update operation (if any) on the object during that transaction. This deferred snapshot option is shown above.
- When a transaction prepares, there is no recovery action required. The saved state cannot be discarded until the final outcome of the transaction is known.
- When a nested transaction commits, the saved recovery state must be propagated to the enclosing transaction. The propagated state may be discarded by the enclosing transaction if an earlier saved state already exists for the enclosing action. If a nested transaction aborts, the recovery state must be used to restore the volatile state of the object.
- When a top–level transaction commits, the saved recovery state can be discarded. As with nested transactions, if a top–level transaction aborts, the recovery state must be used to restore the volatile state of the object.
- If a recoverable object which was created during the course of execution of a top–level transaction does not exist at the end of that transaction, then the recovery state is simply discarded at or before prepare time. The recovery state must be retained until the transaction prepares, in case the object is re–incarnated and reused in the transaction. However, once the transaction prepares, there will be no further introduction of objects into the action group (hence, re–incarnation ceases to be a possibility). If the object does not exist at prepare time, the volatile state cannot possibly be restored so the recovery state can be discarded regardless of the ultimate outcome of the transaction.

Persistence Management

Logically, when a transaction begins, the persistent state of objects which will be used in that transaction is loaded from the stable store. Unless the object is modified in the course of the transaction, there is no further action required for persistence. If an operation modifies the (volatile) state of an object during a transaction, the object must be marked as "modified" so that appropriate steps can be taken when the transaction terminates. When a top–level transaction prepares, persistent objects which have been modified must create a provisional new state in the stable store. For nested transactions, there is no further action for persistence, except to ensure that the object is included in the action group of the enclosing action. Whether a nested transaction commits or aborts, the stable state of objects which were modified during that nested transaction is not known until the enclosing top–level action terminates. The provisional state will become the actual state if the top–level transaction commits. The provisional state can be discarded if the top–level transaction aborts. We treat the case of an object which is prematurely deleted as an "early prepare" for that object. That is, if the object is unmodified, there is no action to take. If the object is modified, the modified state is written provisionally to the stable store where its ultimate fate will be decided at top–level commit or abort just as for all other objects in the action group.

Concurrency Control Management

The concurrency control actions shown in Table 5–1 are just those for strict two–phase locking. The only point of interest is that locks must be preserved if an object leaves extent before the end of transactions in which it was involved. Hence, the lock state must be separate from the object.

As Table 5–1 has shown and the following text has explained, the management of lexically scoped objects in a dynamic transaction environment requires some careful consideration. Although this explanation has been framed in terms of a specific set of recovery, persistence and concurrency control options, there are similar actions to be taken at transaction state transitions for all of the other design choices outlined in the previous chapter.

The remaining design issue which is directly relevant to these choices of mechanism is the relationship of buffer management and recovery techniques and levels of concurrency. This is the final design topic discussed in this chapter.

5.2.4 Buffer Management and Recovery Techniques

The class hierarchy described in Chapter Four offers application programmers a range of mechanisms for achieving persistence, recovery and concurrency control. How should a programmer select among these options? Which combinations make sense? The discussion that follows outlines the principal effects of various choices and offers some guidance for selection.

State-based recovery techniques employ one of two basic mechanisms: *logging* or *shadowing*. In Argus, code inserted by the language processor causes log records to be created whenever changes are made to an "atomic" object [Oki 83]. Guide [Balter *et al* 91] similarly uses deferred updates and logs to support recovery in its transaction model. Aries/NT [Rothermel and Mohan 89] uses a variation of logging called *write-ahead logging* (WAL) in which log records describing changes are forced to the log so that the state of the object need not be flushed to stable storage immediately on transaction commit. System R creates a shadow copy of a page of data just before the first modification to that page in a transaction [Gray *et al* 81]. Updates to the object can then be made "in place". If the transaction aborts, the main version is replaced with the backup copy. Clouds uses shadowing in just the opposite, more pessimistic way: updates made during the transaction are made to the shadow. If the transaction commits, the main copy of the object is replaced with the shadow copy. How were all of these design choices made?

The characteristics of objects which are being traded off in the selection of persistence and recovery mechanisms are flexibility, concurrency and performance. Consider the options for persistence. Persistence can be achieved either by a pure state-based approach or by operation logging. The state-based approach is simple and effective, but may be costly for large objects where the state of the entire object must be copied even when only a small part of the object is being modified. More critically, because the state must be captured consistently, concurrent updates from different transactions cannot be supported [Rothermel and Mohan 89]. Hence, operation logging is the mechanism of choice for large objects, especially if concurrent updates are desirable. However, operation logging in a conventional database system is considerably easier than in an object-oriented system: the only operations supported by database systems are read and write. Hence the operation log needs only a trivial encoding of the operation and the data that was affected by the operation. In an object-oriented system, an operation of some class of objects may have rich semantics and complex effects. The encoding of these operations and their subsequent interpretation during recovery is a considerable engineering undertaking.

Recovery options are similar to persistence options in the sense that state based recovery is simple and effective, but potentially inefficient for large objects. Also, as for persistence, state-based recovery limits concurrency. The use of state-based recovery is limited to objects which employ an exclusive update policy for concurrency control. One further aspect to recovery arises from the fact that some operations are by their nature unrecoverable. For object classes with unrecoverable operations, state-based recovery is not an option. The operation-based recovery class, *CompAction*, was provided specifically to allow class developers to manage unrecoverable operations by compensating actions rather than state restoration.

The general problem of interactions between buffer management and recovery techniques have been extensively addressed in a paper by Haerder and Reuter [Haerder and Reuter 83]. However, since the authors were considering conventional relational databases, some of the unique aspects of object-oriented databases were not fully addressed. The key observation that can be made in light of the discussions of object mechanisms above is that the management of buffers of database pages in conventional systems is just a lower-level manifestation of the constraints on concurrent update. That is, conventional systems do not allow programmers to employ concurrent writer policies, but since the implementation of these systems store multiple objects on a page, they encounter the same kinds of problems with concurrent updates. For performance, conventional systems need to perform concurrent updates on a page even though they do not allow concurrent updates to individual objects on the page. In these implementations, the "object" is a database page and the concurrent update of these "objects" requires the adoption of a non-state-based approach to recovery and persistence. The solutions which they have adopted in their implementation, such as Write-Ahead-Logging, are visible to application programmers in the choice of mechanisms provided through the class hierarchy described in Chapter Four.

The four key design topics of heterogeneity, object lifetimes, commit protocols and nested transactions and buffer management and recovery techniques have been discussed. In each case, there are many possible design choices and many considerations. This discussion has simply attempted to present a plausible set of design tradeoffs that support the goals described in Chapter One for a development environment for reliable distributed applications. The remainder of this chapter describes an implementation of this environment and some experience of its use.

5.3 An Implementation

To validate the ideas put forward in the previous chapters, a prototype implementation has been developed in C++ on a network of Sun workstations. Experience with developing applications using the prototype system provides insight into the practical engineering issues involved in a full-scale development and permit the ideas for reliable distributed applications programming to be put to the test.

The class hierarchy that has been implemented includes all of the classes described in the previous chapter except the log-based persistence class. The components of the execution environment, including machine-to-machine communications via RPC, naming service and object storage, have been implemented more or less as outlined in Chapter Three. RPC is provided by a simple layer on top of IP-based datagram service. Name service is provided by a collection of persistent objects which are accessed and updated under the control of atomic actions. Object storage is provided by the Unix file system, storing one object per file, named with the object UID. The mechanisms used for these services are very similar to the mechanisms used in Arjuna, although the interfaces and organisation are along the lines of the architecture outlined in Chapter Three.

Because the prototype implementation uses the same basic mechanisms for communications and storage as the Arjuna software, the performance of the prototype of this system is consistent with the performance of Arjuna. In both cases, the overhead of dynamic binding, separate handling of locks and other aspects of the class hierarchy structure are completely dominated by network delays and slow access times to the disk through the Unix file system. Although the class hierarchy described here is considerably more complex than the class hierarchy of Arjuna, using multiple inheritance, advanced recovery mechanisms and selective inheritance, the performance aspects of these differences are negligible. The focus of the work described in this thesis has been on separating out the various properties that derive from the distribution and persistence transparencies and experimenting with the use of multiple inheritance as a programming interface for specifying these attributes. As with Arjuna, the performance of subsystems has not been a priority.

There is one substantial difference in potential performance between these two systems: Arjuna uses only state-based recovery; the class hierarchy described here allows for log-based recovery as an alternative to state-based recovery. Not only does log-based recovery allow increased flexibility, for example, by supporting multiple concurrent writers, but for large objects which undergo small changes in the course of a transaction, log-based recovery may exhibit a substantial performance gain over state-based recovery schemes.

5.3.1 Design and Implementation Summary

The design of a development environment for reliable distributed applications has been proven to be feasible by a prototype implementation. Feasibility is a necessary but not sufficient condition - an important question that remains to be answered is whether or not this system actually provides an improved environment for the development of reliable applications. This question can only be answered by experience of developing applications in the environment to determine if they are easier to develop and/or more reliable. The remaining sections of this chapter attempt to address this question. A deeper analysis of the sources of improvement is deferred to the final chapter.

To evaluate the effectiveness of the design requires some measure of subjective judgement - is this a simpler programming interface for reliable distributed applications programming than the alternatives? - and some measure of objective measurement and analysis. The remaining sections of this chapter briefly present a series of example applications, showing the use of the class hierarchy to achieve persistence, recovery and concurrency control in a variety of situations.

5.4 Application Development

Programmers using the class hierarchy must be familiar with object-oriented programming and the general concept of transactions. What other concepts must programmers understand to make use of the system? To answer this question, one must first separate the uses of the system into two different activities:

- designing new classes of objects
- creating and accessing instances of existing classes of objects

The programming interfaces for the two types of use are somewhat different. Each is described with appropriate examples in the sections that follow.

5.4.1 Class design

To design a new class of objects requires attention to the potential uses of the object which in turn determine the properties that class should exhibit. The focus is on the behaviour of a *single* class of application–level objects. In addition to defining the functional behaviour of the class, the class designer must consider:

- whether or not the objects might be persistent
- the extent to which the objects need to be recoverable
- the potential for sharing and the semantics of concurrent access

There is little need to consider transaction boundaries since it may be assumed that, for normal "reliable" applications, all the operations of the class will be invoked from within a transaction. Indeed, the *transaction transparency* property described in Chapter Two requires that the behaviour be independent of whether the operations are invoked from within a transaction or not. However, if a class designer determines that it is necessary or desirable, the implementation of some operations of a class may create nested sub–transactions in which access to other objects may be contained. That is, the implementation of an operation, whether or not it is invoked within the bounds of some enclosing transaction, may itself create new transactions to define the boundaries for persistence, recovery, and concurrency control for its own operations. Bearing in mind the dependencies among transparencies mentioned in the previous chapter, any combination of transparencies can be selected by appropriate use of inheritance. Even access and location transparency can be controlled by the class designer. Access and location transparency will rarely be separated since it makes little sense to have transparent access to an object which cannot be found or to transparently locate an object which cannot be accessed. Recovery, concurrency control and persistence properties are design choices which must be addressed when a class is designed.

5.4.2 Application design

In the development of reliable applications, the role of application designers will be extended to include consideration of appropriate transaction boundaries. At the level of the application, distribution and persistence transparencies can be assumed. Although it is possible for an application to selectively apply these properties to application objects, the modification of class properties casts the application designer in the role of class designer. For most application designers, the set of object classes and their properties will be pre-determined.

To the extent that designing an application involves defining new classes of objects to implement the application, the two roles of “class designer” and “applications programmer” may be assumed by a single person. However, in the design of a single class, the developer may consider all other classes of objects to exhibit the uniform transparent behaviour described above. Hence the modular, encapsulated design feature of object-oriented design is carried through to include the persistence and distribution transparencies. In encapsulating reliability, distribution and persistence issues within the programming language’s unit of modularity, the object class, the programmer gains conceptual simplicity which should improve his ability to produce reliable systems.

5.4.3 Application Example 1 - A Bank Account

A classic example of a transaction processing application, the management of bank accounts, is a good example with which to illustrate the kinds of objects that require a full range of transparency support including persistence, concurrency control, recovery, possibly distribution (access and location transparency) and possibly replication. This first example demonstrates the capability of the class hierarchy described above to model conventional transaction processing problems and illustrates the usefulness of inheritance in modelling complex real-world problems.

Consider the abstraction of (a simplified view of) an account at a bank. The account consists of a balance which is positive for deposit accounts, negative for loan accounts and varies from positive to negative for chequing accounts. There are operations on the account – payments into the account, withdrawals from the account and interest accumulated to the balance. An example of a C++ definition of the class, BankAccount is given in Figure 5-2.

```
class BankAccount : public StateRecoverable,
                    public StatePersistent,
                    public Shared {
public:
    BankAccount();
    void Deposit(Money amount);
    boolean Withdraw(Money amount);  // returns true if succeeded
    Money Balance(void);
    void AccrueInterest(float rate);
protected:
    // Two operations inherited from CheckpointObject
    ErrorCode SaveState (ObjectState &);
    ErrorCode RestoreState (ObjectState &);
private:
    Money balance;
};
```

Figure 5-2: A C++ definition of the abstract class BankAccount

Some specific types of account which a bank offers can be derived from this basic bank account class as shown in Figure 5-3.

```
class DepositAccount : public BankAccount {
public:
    DepositAccount();
};

class ChequingAccount : public BankAccount {
public:
    ChequingAccount (Money OverdraftAllowance);
    boolean Withdraw(Money amount);  // returns true if succeeded
    void AccrueInterest(float rate);
protected:
    ErrorCode SaveState (ObjectState &);
    ErrorCode RestoreState (ObjectState &);
private:
    Money OverdraftLimit;
};

class LoanAccount : public BankAccount {
public:
    LoanAccount (Money amountOfLoan);
};
```

Figure 5-3: Specialisations of the BankAccount class

For each type of account, the desired transparencies of concurrency control, recovery and persistence are inherited from the property classes Shared, StateRecoverable and StatePersistent respectively. Client applications accessing these accounts from within the boundaries specified by transactions will be assured of these transparency properties. Because persistence requires a notion of identity, these objects will inherit from the class Identified and will thus be locatable through a distributed location or naming service. The bank accounts may be transparently distributed over a network.

Because the BankAccount class inherits persistence and recovery properties, the class developer needs to implement the SaveState and RestoreState operations inherited from CheckpointObject. These operations will save the state of the bank account object as and when necessary for recovery or persistence. The only other additional effort required on the part of the class developer to achieve the transparencies is to add invocations of appropriate lock operations to the operations of the class BankAccount. A sample of the resulting code to implement the operations of the basic BankAccount class is given in Figure 5–4.

```
#include "BankAccount.h"

BankAccount::BankAccount()  { balance = 0; }

void BankAccount::Deposit(Money Amount)
{
    TakeLock(new Lock(WriteLock));
    balance += Amount;
}

boolean BankAccount::Withdraw(Money Amount)
{
    TakeLock(new Lock(WriteLock));
    balance -= Amount;
    return true;
}

Money BankAccount::CurrentBalance(void)
{
    TakeLock(new Lock(ReadLock));
    return balance;
}

BankAccount::AccrueInterest(float rate)
{
    TakeLock(new Lock(WriteLock));
    balance += balance * rate;
}

ErrorCode BankAccount::SaveState (ObjectState &SaveBuffer)
{
    StateRecoverable::SaveState(SaveBuffer);
    StatePersistent::SaveState(SaveBuffer);
    Shared::SaveState(SaveBuffer);
    return SaveBuffer.Pack(balance);
}

ErrorCode BankAccount::RestoreState (ObjectState &RestoreBuffer)
{
    StateRecoverable::RestoreState(RestoreBuffer);
    StatePersistent::RestoreState(RestoreBuffer);
    Shared::RestoreState(RestoreBuffer);
    return RestoreBuffer.Unpack(balance);
}
```

Figure 5–4: Implementation of the class BankAccount

Derived classes such as ChequingAccount need not specify transparency properties again since these are inherited from the parent class BankAccount.

Selected operations might be overloaded in the derived class to produce the appropriate semantics for that type of account. For example, the ChequingAccount class might maintain overdraftAllowance as a private data member and overload the operations:

- Withdraw, to check against the allowed overdraft
- AccrueInterest, to prevent the account accruing interest
- SaveState and RestoreState to save and restore the private data associated with this type of account

Sample code for these operations is shown in Figure 5-5. The other derived classes would be similarly implemented.

```
ChequingAccount::ChequingAccount (Money Overdraft)
{
    OverdraftLimit = Overdraft;
}

boolean ChequingAccount::Withdraw (Money Amount)
{
    if ((Balance() + Amount) > OverdraftLimit) {
        BankAccount::Withdraw(Amount);
        return true;
    }
    return false;  // Not enough money in account
}

void ChequingAccount::AccrueInterest (float rate)
{
    error ("This account does not pay interest");
}

ErrorCode ChequingAccount::SaveState (ObjectState &Buffer)
{
    BankAccount::SaveState(Buffer);
    return Buffer.Pack(OverdraftLimit);
}

ErrorCode ChequingAccount::RestoreState (ObjectState &Buffer)
{
    BankAccount::RestoreState(Buffer);
    return Buffer.Unpack (OverdraftLimit);
}
```

Figure 5-5: Overloading selected operations in a derived class

A client operation making use of a bank account object would normally (dynamically) enclose the operation invocations on the bank account in a transaction as discussed in Chapter Three. Figure 5–6 illustrates the use of transactions in a simplified cheque clearing operation in which it is assumed that an accounting log is updated with the status of the operation regardless of whether the cheque cleared. The log is updated atomically with the withdrawal to ensure that the account balances agree with the recorded operations even in the event of a failure during the cheque clearing operation such as a system crash.

```
void ClearCheque(ChequingAccount &Acct, Money Amount, int Number)
{
    Transaction Action;

    Action.Begin();

    if (Acct.Withdraw(Amount))
        AccountLog.Record (Acct, Amount, CLEARED, Number);
    else
        AccountLog.Record (Acct, Amount, BOUNCED, Number);

    Action.Commit();
}
```

Figure 5–6: A Client accessing a BankAccount from within a Transaction

The overview presented here of a simplified banking example is intended to show the application of the classes described in the previous chapter to a traditional transaction processing problem – an all or nothing, recoverable, concurrency controlled, persistent atomic transaction example. The more general and powerful use of these properties, the *selective* application of properties to object classes, is illustrated in the examples that follow.

5.4.4 Example 2 – A Producer–Consumer Queue

Producer–consumer queues are somewhat problematical entities for traditional transaction systems because it is so difficult to reconcile the recovery property of transactions with the need for concurrent updates implied by the use of a producer–consumer queue. If the enqueue actions of the producer are to be recoverable in the event of a transaction abort then the corresponding dequeue actions and all the effects generated from them must be included in the transaction of the producer. However, such an all encompassing transaction restricts (to one) the number of producers that can simultaneously access the queue. Indeed, even with a single producer and a single consumer, unless the entire history of enqueuing and dequeuing operations is enclosed in a single transaction, some unrecoverable behaviour may occur. For many applications however, although concurrency control is a necessity, to ensure that consumers get consistent units of data from the queue, overall recovery may not be necessary. Dequeued objects which were produced by producers that later failed may have no adverse effect on the system and hence may not need to be 'recovered from'.

For example, suppose that as part of some real-time control or tracking problem, several sensor units are feeding a single evaluator through a multi-producer, single-consumer queue (see Figure 5-7). Assume that critical real-time signal processing, such as Kalman filtering, is performed by each unit before sending higher-level items into the queue. Also, assume that the sensor units are fail-silent, implying that the failure of a unit does not invalidate the data which was previously received from it. Hence, the evaluator does not need recovery actions to eliminate data received from a failed producer. Of course, higher-level compensation actions may be required to account for the loss of the producer - for example, sending control signals to other sensors to increase their range of coverage.

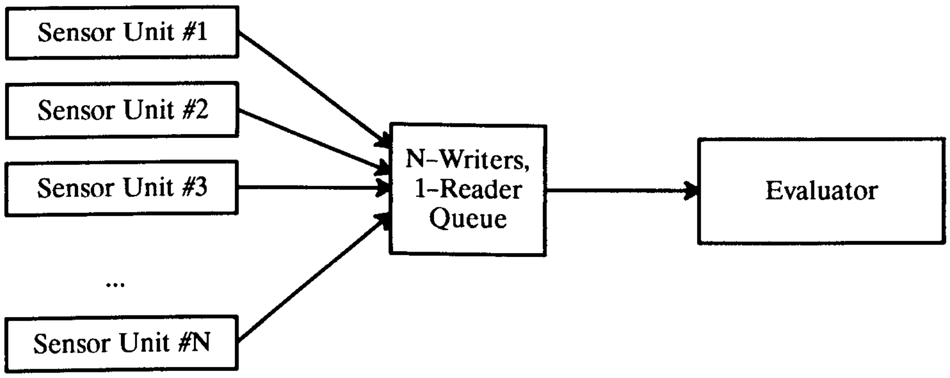


Figure 5-7: Multiple signal sources merging through a single queue

The queue object in this application needs concurrency control, but not persistence or recovery. (Another kind of application such as distributed simulation might require recoverable and persistent queues). Using the classes defined in the previous chapter, one could define the queue class as shown in Figure 5-8.

```
class SignalQueue : public Shared {
public:
    SignalQueue(void);
    void Enqueue(Signal);
    Signal Dequeue (void);
    boolean IsEmpty (void);
};
```

Figure 5–8: Class definition for N–writer, 1–reader, unrecoverable queue

A simple application of the Shared class described in the previous chapter would severely restrict concurrency. Once a single transaction had updated the queue, acquiring a write lock on critical data structures, no other transaction could access those data structures at all until the first transaction had terminated. Since the SignalQueue requires an unconventional multiple–writer, single–reader concurrency control policy, the application designer might want to provide a non–locking concurrency controller based for example on timestamps. Alternatively, keeping the lock–based concurrency controller, an application designer might simply define a new lock type with the appropriate conflict checking policy. In the latter case, the implementation of the class SignalQueue would employ the modified lock type in the implementation of the individual operations, Enqueue, Dequeue and IsEmpty.

5.4.5 Example 3 - A Network Routing Database

A network routing database could be an example of an application object that requires concurrency control and recovery, but not persistence. Consider a communications system in which routers operate as follows:

- When a router object is first created, it acquires a consistent snapshot of the network topology from a neighbouring node.
- As nodes go up and down and communications connectivity changes, routing table updates are broadcast to all routers by the routers that detect the changes.
- An individual update message may modify several parts of the routing table and must be considered atomic.
- Since updates are circulated and processed asynchronously, it is possible for two updates to be concurrently applied to the same routing table.

- Because the network topology and connectivity are so volatile (in this example), individual router objects do not attempt to maintain their state across machine crashes – it would be grossly out-of-date anyway when the node recovered. Instead, this kind of recovery from catastrophic failure is handled at a higher level, by using a snapshot of routing data from some surviving node, or by re-computing the routing information from some other method e.g., manual intervention or broadcast to all nodes requesting routing and topology information.

```
class RoutingTable :     public Shared,
                         public StateRecoverable {
public:
    RoutineTable();
    void    AddEntry (Location &src, Location &dst);
    void    ProcessUpdates (EntryList &l);
    boolean DirectConnection (Location &from, Location &to);
    Route   FindRoute (Location &from, Location &to);
protected:
    ErrorCode SaveState (ObjectState &) const;
    ErrorCode RestoreState (ObjectState &);
private:
    RouteTable Table;
};
```

Figure 5–9: Class Definition for a Recoverable, Concurrency-controlled Routing Table

The class definition for RoutingTable shows the selective use of the "property classes" StateRecoverable and Shared. The operations SaveState and RestoreState must be provided by the class developer to support state-based recovery. Standard locking policies (strict 2-phase locking with single-write XOR multiple-reader locks) seem appropriate so the standard lock class can be used without modification. As in the previous example, the implementations of the operations must take appropriate locks before performing operations. This simplified example could be made more realistic by the addition of new lock modes such as intention-read and intention-write to support multi-granular locking for the RouteTable aggregate. The object-oriented approach makes this particularly easy since the class developer would need only to derive a new type of lock that provided the new locking modes and implemented the operation to compute the conflict matrix including those new modes.

5.4.6 Example 4 - A Dynamic Storage Allocator

Another example of a class of objects that is naturally concurrency-controlled and recoverable, but not persistent is the class of objects used to implement a dynamic storage allocator. Because dynamic (heap) storage is normally volatile, the data structures that track allocation and deallocation need only to be volatile. However, concurrency control and recovery may still be necessary. Because successive allocations are independent and there are few read-only accesses to the data structures, concurrency control may be provided by a a simple semaphore, locking out all access during any operation. However, such a solution does not address the requirement for recovery. For example, if a transaction aborts in the course of execution of an operation invocation, how will the dynamic storage allocated before the abortion be deallocated? If the allocator's data structures are recoverable, the abortion of the transaction will undo the effects of the allocation, freeing the storage for other use.

The state-based recovery scheme offered by the StateRecoverable class is not valid in the presence of multiple concurrent updates. Hence, an operation based recovery scheme is called for. As described in Chapter Four, a class developer who derives an application class from an OperationRecoverable class, such as UndoRecoverable, must invoke the AddOperation operation, passing an undo operation (closure), whenever an application-level operation is invoked. The UndoRecoverable class logs the undo operations as the transaction progresses. If the transaction should abort, the action manager of the UndoRecoverable class will apply the logged undo operations. If a transaction terminates successfully (commits), the logged undo operations can be discarded.

Figure 5-10 illustrates the use of the UndoRecoverable class to define a recoverable storage allocator class. Concurrency control for this class of objects will be provided by simple binary semaphore, so there is no need to inherit the lock-based class, Shared. In this example, it is assumed that storage is allocated according to some simple allocation scheme such as "first-fit" [Knuth 73].

```
class HeapAllocator : public UndoRecoverable {
public:
    HeapAllocator();
    void *Allocate (unsigned int nbytes);
    void Deallocate (void *);
private:
    BinarySemaphore HeapLock;
    Chunk *FreeList, *AllocatedList;
};
```

Figure 5–10: Defining a Recoverable Storage Allocator

The allocate operation finds an appropriate sized chunk of storage on a “free list” and moves it to the “allocated list” before returning it to the caller. The deallocate operation simply moves the storage chunk back to the free list. Undo operations for allocate and deallocate operations are straightforward and idempotent. The “undo objects” represent closures of behaviour and state which can be applied to reverse the effects of a previous operation. The actual first–fit algorithm is provided by utility routines alloc and dealloc which are not shown.

```
struct Chunk { int size; Chunk *next; int bytes; };

//
// Define some undo operations for the allocator
//
class MoveToFreeList : public Operation {
public:
    MoveToFreeList(Chunk *ptr);
    ErrorCode Perform();
private:
    Chunk *p;
};
class MoveToAllocatedList : public Operation {
public:
    MoveToAllocatedList(Chunk *ptr);
    ErrorCode Perform();
private:
    Chunk *p;
};

//
// Implement the allocator operations
//
HeapAllocator::HeapAllocator()
{
    HeapLock.V();
}

void *HeapAllocator::Allocate(unsigned int nbytes)
{
    void *ptr = 0;
    HeapLock.P();
    ptr = alloc(nbytes+sizeofheader);
    if (ptr)
        AddOperation (new MoveToFreeList(ptr));
    HeapLock.V();
    return &ptr.bytes;
}

void HeapAllocator::Deallocate (void *ptr)
{
    Chunk *c = GetChunkFromPtr(ptr);
    HeapLock.P();
    AddOperation (MoveToAllocatedList(c));
    dealloc(c);
    HeapLock.V();
}
```

Figure 5-11: Implementing a Recoverable Storage Allocator

The recovery property of the HeapAllocator class is providing tolerance to transaction aborts which may be due in turn to programmer initiated aborts, communications failures, deadlocks, or other causes of transaction failure. The class as defined does not cope with asynchronous faults since, for example, no provision is made for releasing the semaphore if the Allocate operation should be interrupted by a fault which later leads to transaction failure. Asynchronous faults can be dealt with in the same manner as shown above, although the code becomes considerably more complex and has been omitted since it adds no insight to the usefulness of selective transparency in this example.

5.4.7 Example 5 - A Printing Subsystem

This final example illustrates the use of a variety of properties for the various components of a printing subsystem in a modern operating system. The users of any modern operating system expect to be able to queue files to be printed. Furthermore, they expect that the files are copied into a spool area before printing and that the queue of files waiting to print can be inspected. Finally, users expect to be able to abort the printing of some previously spooled file whether it is still in the queue or actively printing. From this informal statement of requirements, we can develop an object-oriented design for such a subsystem. There must be a Spooler class which is the user's interface to the printing subsystem. The Spooler class must support the operations: Print, Cancel and ShowQueue. There must be a SpoolDirectory into which the files are copied before printing and a PrintQueue which keeps track of the order of printing of the files, accounting information etc. Finally, there must be a PrintDeviceDriver which transfers data from the files in the SpoolDirectory to the actual printing device in an order determined by the PrintQueue. This example does not treat the problems of security of the files and the queue or the issues of accounting for page charges. These classes are depicted in Figure 5-12.

The class Spooler provides the following operations:

- Print - allocates a new job number, copies the file to SpoolDirectory and adds an entry for it in PrintQueue
- Cancel - removes the specified job from SpoolDirectory and PrintQueue
- List - enumerates the entries in PrintQueue

The allocation of a job number is performed under control of a mutual exclusion lock, but since the number has no meaning other than to be a locally unique identifier, more sophisticated, transaction-level locking is not required. Concurrent access is permitted and uncontrolled. Cancelling involves invoking operations on other objects (PrintQueue and SpoolDirectory), but does not affect the nextjobnumber. List is a readonly operation. The class Spooler inherits from class CompRecoverable so that a Print request that is made during a transaction that later aborts is automatically cancelled. The recovery method is compensating rather than "undo" since the job cannot be "unprinted" if it has already left the spool directory / print queue. The effect of a transaction abort will be to remove any as-yet-unprinted files from the queue and directory.

The class PrintQueue provides the following operations:

- Add – adds a new job to the end of the queue, recording the job number, original file name and user name.
- Remove – removes a job from the queue if present.
- List – enumerates the entries in the queue in order.

PrintQueue objects are persistent, so that system crashes do not lose track of the jobs that have been queued to print. Printing resumes after a crash by restarting the printing of the first job in the queue. These objects must be recoverable so that transaction aborts cause queue entries made during those transactions to be removed. The recovery method of choice is "UndoRecoverable" in order to permit concurrent updates to the queue. However, concurrent updates to individual entries must be controlled, hence the class PrintQueue inherits concurrency control from the class Shared. This is an example of a class that might employ a derivative lock type to permit concurrent update access to the queue object, but control concurrent access to the individual entries. The implementations of these operations are mostly self-evident, but the Remove operation will also invoke PrintDeviceDriver.Abort if the first job in the queue is removed.

The class SpoolDirectory provides the following operations:

- CopyFile – copies the specified file into the spool area
- DeleteFile – deletes the specified file if it exists in the spool area

The class SpoolDirectory inherits the same properties as the class PrintQueue and for most of the same reasons: persistence to survive system crashes, undo-based recovery to cope with transaction aborts yet allow limited concurrency, and concurrency control, possibly based on a derivative lock class which would allow concurrent update of the spool area, but not of individual entries in it.

The class PrintDeviceDriver provides the following operations:

- StartPrint - begins a background activity of transferring data from the specified file to the printing device.
- Abort - stops printing

It is assumed that the actual transfer of lines of data to the printer is carried out in the background, perhaps by having the printer interrupt at the end of each line or page of data printed. That interrupt could trigger the transfer of the next chunk of data to the printer and so on until the entire file was printed. When all data has been transferred, the PrintDeviceDriver object invokes PrintQueue.Remove to delete the first element of the PrintQueue.

Additional operations must be provided to read the data from the files in the spool area and to synchronise the starting and stopping of the printer as the PrintQueue transitions from empty to non-empty and back. However, these operations do not contribute anything to the understanding of the use of the property classes and have been omitted from the example.

```
class Spooler : public virtual CompRecoverable {
public:
      int  Print(const char * filename);
      void Cancel(int job);
      void ShowQueue();
private:
      mutex lock;
      int nextjobnumber;
      PrintQueue Queue;
      SpoolDirectory Dir;
};

class PrintQueue : public virtual LogPersistent,
                   public virtual UndoRecoverable,
                   public virtual Shared {
public:
      void Add(int job, const char *filename, const char *user);
      void Remove(int job);
      void List();
};

class SpoolDirectory :   public virtual LogPersistent,
                         public virtual UndoRecoverable,
                         public virtual Shared {
public:
      void CopyFile(int job, const char *filename);
      void DeleteFile (int job);
};

class PrintDeviceDriver {
public:
      void StartPrint(int job, const char *filename);
      void Abort();
private:
      int currentjob;
};
```

Figure 5-12: Classes of a Printing Subsystem

5.4.8 Summary of Examples

The examples given in this chapter have illustrated the use of the class hierarchy for selective application of distribution and persistence transparencies. Many combinations of transparencies have not been illustrated here although they may be useful in certain applications. The ability to selectively apply transparencies through inheritance and to derive new property classes, e.g., operation-based recovery vs. state-based recovery, are principal strengths of the object-oriented approach to reliable distributed programming. Although the examples given here use C++ to illustrate the concepts, the technique is applicable to any object-oriented language that supports multiple inheritance.

This chapter began with a brief discussion of some key design issues in the development of the property classes and finished with a series of examples illustrating the usefulness of the class hierarchy. This progression has illustrated both the feasibility and the value of selective transparency in general and specifically the system described in Chapters Three, Four and Five. The next and final chapter summarises this thesis by reviewing the goals established in the first chapter and evaluating the software solution to address those goals which was developed in the following chapters.

6 Conclusions and Further Work

"The successful construction of all machinery depends on the perfection of the tools employed, and whoever is a master in the art of tool-making possesses the key to the construction of all machines"
Charles Babbage [Babbage 1889]

The development of reliable computer software and systems is gaining importance from two sources: market pull and technology push. Market pull comes from the fact that computers are being used in ever more demanding applications including life-critical applications. Hence, people are becoming increasingly reliant on the correct functioning of computer systems. Technology push arises from the critical mass in reliability technology which has been developed over the past several decades and the dramatic increases in computer hardware reliability and performance that make the application of this reliability technology economically viable. Software developers must take steps to improve the reliability of their systems to meet the reliability requirements of applications.

Improvements in the reliability of software systems have not kept pace with increases in complexity of these systems. Despite the increased awareness of the importance of software quality, software applications today continue to be delivered late and error-laden. The source of this unfortunate problem is not difficult to discover. The ambitions of software developers have outstripped their ability to deliver reliable software. The complexity and plasticity of large software systems requires a powerful new approach to reliability. Since the technology to deliver error-free applications is not available, large software systems must be designed to accommodate the inevitable faults in software design and implementation as well as (equally inevitable) hardware faults.

The technological push that arises from the availability of distributed systems adds urgency to the problems of developing reliable software. Data and code distribution, management of communications and system failures, and access to remote data have been added to the already difficult task of application programming. To address these problems in the development of reliable distributed applications, a new approach is required. This thesis proposes a program development environment, based on objects and actions, that provides a powerful, flexible tool for the development of reliable distributed applications. This chapter identifies some of the remaining issues for further research and summarises the major contributions of the thesis.

6.1 Some Remaining Issues

"When the solution that we have finally obtained is long and involved, we naturally suspect that there is some clearer and less roundabout solution: *Can you derive the result differently? Can you see it at a glance?*" [Polya 57, p. 61]

Further research is presently underway to extend the class hierarchy to include replication and migration as abstract "property" classes. These extensions may take the same form as the existing property classes and use the same mechanisms. Depending on the choice of replication protocol, replication support may also require a change in the interface to the underlying support systems such as RPC and storage services. A variety of other open design questions are discussed in the sections that follow.

6.1.1 Open Design Issues

There are many aspects to *engineering* a production quality distributed transaction system which have been deliberately ignored in the prototype developed for this thesis as they convey no new research content and would only clutter the discussion of key issues. However, the following list enumerates some of the principal design embellishments which would be required to deliver a production quality system.

- Commit Protocol – The prototype implementation employs only a two–phase commit protocol. In practice, two–phase commit suffers from blocking problems, when the co-ordinator node crashes, and possibly some reduced concurrency, due to the late release of locks. More powerful commit protocols such as Open Commit Protocol [Rothermel and Pappe 89] [Rothermel and Pappe 90] and three–phase commit [Skeen 81] could be introduced without affecting the substance of this thesis.
- Administration – Large distributed systems are rarely controlled by a single administrative body. Facilities for system management that take account of different administrative domains of control are vitally necessary to successful distributed systems deployment.
- Deadlock – Deadlock has been treated in this system by timeouts, causing an aborted transaction whenever a lock cannot be acquired "in time". This simplistic deadlock management technique may well suffice even in a large system since, "Measurements of experimental and commercial DBSs indicate that deadlocks are much rarer than conflicts" [Bernstein *et al* 87, p. 88]. However, a variety of more sophisticated techniques for deadlock detection, prevention and recovery could be applied [Bernstein *et al* 87].
- Transactions on objects of arbitrary type may cause difficulties in precise definition of semantics [Jacobsen 82].
- Alternative concurrency control policies based on timestamping could be added either independent of the lock–based policies or integrated with them via some kind of abstract concurrency control class.

6.1.2 Dependencies on Commit Protocol

A complex issue which needs further thought is how to modify the existing hierarchy to incorporate different commit protocols. While the specific locking protocol (e.g., two–phase locking) is nicely encapsulated in the implementation of the *Shared* class, the commit protocol (e.g., two–phase commit) is visible in the *interface* to the object managers. This implies that a different atomic action object (employing a different commit protocol e.g., three–phase commit) would require a different implementation of the classes. Yet a persistent object ought to be able to participate (even simultaneously) in atomic actions employing different commit protocols. This is not possible in the current design.

6.2 Some Observations of the use of C+ +

The prototype software developed for validating the ideas presented in this thesis was written entirely in C+ +. C+ + is a hybrid object-oriented language, developed as an extension of C (of which it is a proper superset). The goals of the C+ + language designers did not include the development of distributed applications and it shows.

C+ + adopts the same shared memory, single thread-of-control model of C. This programming model is difficult to map to a distributed system where objects may reside in multiple, physically and logically distinct address spaces and parallel execution is not only possible, but inevitable. This mismatch between the programming model presented by C+ + and the goals of this thesis was the principal source of difficulty in the development of the system.

A further complication arises from the lack of support for "reflection" in C+ +. There is no support in the C+ + language for program access to meta-data such as run-time type identification, dynamic instantiation, or type structure enquires. These missing facilities require further restrictions for programmers in the subset of C+ + which can be accommodated in the prototype system. See [Parrington 90] for a detailed discussion of the problems.

In summary, C+ + is an excellent object-oriented programming language for non-distributed programming despite certain deficiencies in current versions of the language, such as the lack of garbage collection, no safe downcasting (due to the lack of run-time type information), no exception handling facilities. Some of these problems are being addressed in the C+ + standardization work which is underway and it is only a matter of time before these problems are solved. The object-oriented facilities of the language served well in the development of the prototype system described in this thesis. However, C+ + is an unfortunate choice for distributed systems development. The irregularities of the language, manual storage management, complex interactions of scope and extent, and fundamental shared memory model are all obstructions which are difficult to overcome without substantial and continuing effort of the part of programmers.

6.3 Thesis summary

This thesis has presented a novel architecture for distributed transaction processing in which the management of persistence, provision of transaction properties, and organisation of support services are all gathered into a unified design based on the object-oriented programming model. Derived from the pioneering work of the Arjuna project, this thesis takes the idea of using inheritance to control the application of properties much further than Arjuna, separating out the protocols required to support each property and providing new flexibility, through multiple inheritance.

Isolating properties in separate classes which can be individually inherited clearly establishes the protocols required by each property and identifies interdependencies between properties. This exposition of protocols for properties such as persistence, recovery and concurrency control and the analysis of the interactions of these properties are contributions of this thesis that will assist developers of transaction-based software whether or not they choose to use inheritance as the mechanism for conveying properties.

The integration of object-oriented programming concepts, persistent data as a language feature, and transaction processing mechanisms provides increased semantic information, better isolation of locking protocols and commit protocols, and independent management of persistence, recovery, concurrency, replication and migration. Each of these properties contributes to the reduction of complexity for programmers developing reliable distributed applications. The use of multiple inheritance allows programmers to achieve unprecedented flexibility and selectivity in the application of distribution and persistence transparencies to application objects.

To fully exploit the power of this class hierarchy requires consideration of the overall system architecture – not only the structure of the transaction system itself and the programming language interface to the system, but also the relationships between the transaction system (here embodied in a class hierarchy) and operating systems, networks and storage subsystems. Chapter Three placed the transaction services component in context by describing a system architecture that includes the wider issues of naming, binding, communications and storage in a distributed system.

A prototype system has been implemented, verifying the feasibility of both the architectural organisation described in Chapter Three and the approach of providing transaction services by the use of multiple inheritance in an object-oriented programming language. Chapter Five explained some of the detailed design and implementation issues that had to be addressed to implement the proposed system and illustrated the ways in which it could be exploited to provide reliable applications in a wide variety of domains with differing requirements.

The development of reliable distributed applications is a complex task which must be performed successfully to meet the demands of modern software applications. This thesis makes a contribution to the field of distributed transaction processing and databases, explaining how to apply the latest developments in programming languages to advantage. The thesis also makes a contribution to the field of object-oriented analysis, design, and programming in investigating the power and limitations of current object-oriented programming models.

References

[Abrial 74]

J. R. Abrial, "Data Semantics", in Data Base Management, J. W. Klimbie and K. L. Kofferman, (eds.), North-Holland Publishing Co., New York, 1974.

[Agesen *et al* 89]

O. Agesen, S. Frølund, M. H. Olsen, "Persistent and Shared Objects in Beta", Technical Report DAIMI IR - 89, Computer Science Department, Aarhus University, DK, April 1989.

[Albano *et al* 88]

A. Albano, G. Ghelli, R. Orsini, "The Implementation of Galileo's Persistent Values", in Data Types and Persistence, M. P. Atkinson, P. Buneman, R. Morrison (eds.), Springer-Verlag, 1988, pp. 253-263.

[Allchin 83]

J. E. Allchin, "An Architecture for Reliable Decentralized Systems", Ph.D. Thesis, School of Information and Computer Science, Georgia Institute of Technology, September 1983. Also available as Technical Report GIT-ICS-83/23.

[Allchin and McKendry 83]

J. E. Allchin and M. S. McKendry, "Synchronizing and Recovery of Actions", *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, August 1983, pp. 31-44.

[Anderson and Kerr 76]

Anderson, T and R. Kerr, "Recovery Blocks in Action: A System Supporting High Reliability", *Proceedings of 2nd International Conference on Software Engineering*, San Francisco, October 1976, pp. 447-457.

[ANSA 89]

Advanced Networked Systems Architecture (ANSA) Reference Manual, Volume A, Release 1.00, Part VI, Computational Projection, March 1989.

[Atkinson *et al* 81]

M. P. Atkinson, K. J. Chisholm and W. P. Cockshott, "PS-Algol: An Algol with a Persistent Help", ACM SIGPLAN Notices, Vol. 17, No. 7, July 1981, pp. 24-31.

[Atkinson *et al* 88]

M. P. Atkinson, P. O. Buneman, R. Morrison, Data Types and Persistence, Springer–Verlag, Berlin, 1988.

[Atkinson *et al* 90]

M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison, "An Approach to Persistent Programming" in Readings in Object–Oriented Database Systems, S. B. Zdonik and D. Maier (eds.), Morgan Kaufmann, 1990, pp. 141–146.

[Avizienis 85]

Avizienis, A, "The N–Version Approach to Fault–Tolerant Software", IEEE Transactions on Software Engineering, Vol. SE–11, No. 12, December 1985, pp. 1491–1501.

[Babaoglu 90]

O. Babaoglu, "Fault–Tolerant Computing based on Mach", Operating Systems Review, Vol. 24, No. 1, January 1990, pp. 27–39.

[Babbage 1822]

C. Babbage, "On the Application of Machinery to the Purpose of Calculating and Printing Mathematical Tables", A Letter to Sir Humphry Davy, Bart., President of the Royal Society, July 3, 1822, in *Charles Babbage and his* Calculating Engines, P. Morrison and E. Morrison, (eds.), Dover Publications, New York, 1961, pp. 298–305.

[Babbage 1889]

C. Babbage, "Chapter XIII, *The Exposition of 1851*", in Calculating Engines, H. P. Babbage (ed.), E. and F. N. Spon, 1889, and reprinted in *Charles Babbage and his* Calculating Engines, P. Morrison and E. Morrison, (eds.), Dover Publications, New York, 1961, pp. 322–330.

[Bailey 89]

P. J. Bailey, "Performance Evaluation in a Persistent Object System", *Proceedings of the Third International Wokshop on Persistent Object Systems*, 1989, pp. 373–385.

[Balter *et al* 91]

R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. L. Dot, H. N. Van, E. Paire, M. Rivelli, C. Roisin, X. R. de Pina, R. Scioville, and G. Vandôme, "Architecture and Implementation of Guide, an Object-Oriented Distributed System", Computing Systems, Vol. 4, No. 1, Winter 1991, pp. 31–68.

[Banâtre *et al* 83]

J. P. Banâtre, M. Banâtre, and F. Ployette, "Construction of a Distributed System Supporting Atomic Transactions", *Proceedings of the 3rd Symposium on Reliability in Distributed Software and Database Systems*, IEEE, October 1983, pp. 95–99.

[Banâtre *et al* 86]

J. P. Banâtre, M. Banâtre, and F. Ployette, "An Overview of the Gothic Operating System", Rapport de Recherche 504, INRIA, March 1986.

[Barak and Kornatzky 87]

A. Barak and Y. Kornatzky, "Design Principles of Operating Systems for Large Scale Multicomputers", in Experiences with Distributed Systems, Lecture Notes in Computer Science, Vol. 309, Springer–Verlag, Berlin, 1987, pp. 104–123.

[Bennett 90]

J. K. Bennett, "Experience with Distributed Smalltalk", Software - Practice and Experience, Vol. 20, No. 2, February 1990, pp. 157–180.

[Bernstein and Goodman 81]

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

[Bernstein *et al* 87]

P.A. Bernstein, V. Hadzilacos and N. Goodman, Concurrency Control and Recovery in Database Systems, Addison–Wesley, 1987.

[Bershad *et al* 87]

B. N. Bershad, D. T. Ching, E. D. Lazowska, J. Sanislo, M. Schwartz, "A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems", IEEE Transactions on Software Engineering, Vol. SE–13, No. 8, August 1987, pp. 880–894.

[Birman and Joseph 87]

K. Birman and T. Joseph, "Exploiting Virtual Synchrony in Distributed Systems", *Proceedings of the 11th Symposium on Operating System Principles*, ACM SIGOPS, November 1987, pp. 123–138.

[Birman *et al* 88]

K. Birman, T. Joseph and F. Schmuck, "ISIS – A Distributed Programming User's Guide and Reference Manual", The ISIS Project, Department of Computer Science, Cornell University, Ithaca, NY, 14853, March 1988.

[Birrell and Nelson 84]

A. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls", ACM Transactions on Computer Systems, Vol. 2, No. 1, February 1984, pp. 39–59.

[Birtwhistle *et al* 73]

G. M. Birtwhistle, O–J. Dahl, B. Myhrhaug and K. Nygaard, Simula Begin, Academic Press, 1973.

[Black *et al* 87]

A. Black, N. Hutchinson, E. Jul, H. Levy, L. Carter, "Distribution and Abstract Types in Emerald", IEEE Transactions on Software Engineering, Vol. SE–13, No. 1, January 1987, pp. 65–76.

[Brown 87]

A. L. Brown, "A Distributed Stable Store", *Proceedings of the Second International Workshop on Persistent Object Stores*, Appin Scotland, 1987.

[Brown 89]

A. L. Brown, "Persistent Object Stores", PhD Thesis, University of St Andrews, 1989.

[Browne *et al* 83]

J. C. Browne, J. E. Dutton, V. Fernandes, A. Palmer, A. R. Tripathi, P. Wang, "Zeus: An Object-Oriented Distributed Operating System for Reliable Applications", RADC Technical Report, October 1983.

[Campbell and Habermann 74]

R. H. Campbell and A. N. Habermann, "The Specification of Process Synchronization by Path Expressions", Lecture Notes in Computer Science, Vol. 16, Springer-Verlag, 1974.

[Campbell and Madany 91]

R. H. Campbell and P. W. Madany, "Considerations of Persistence and Security in Choices, an Object-Oriented Operating System", Technical Report UIUCDCS-R-91-1670, Department of Computer Science, University of Illinois at Urbana-Champaign, March 1991.

[Campbell *et al* 89]

R. H. Campbell, G. M. Johnston, P. W. Madany, and V. F. Russo, "Principles of Object-Oriented Operating System Design", Technical Report UIUCDCS-R-89-1510, Department of Computer Science, University of Illinois at Urbana-Champaign, April 1989.

[Chase *et al* 89]

J. S. Chase, F. Amador, E. Lazowska, H. Levy, R. Littlefield, "The Amber System: Parallel Programming on a Network of Multiprocessors", *Proceedings of the 12th ACM Symposium on Operating System Principles*, Litchfield Park AZ, December 1989, pp. 147–158.

[Codd 79]

E. F. Codd, "Extending the Database Relational Model to Capture More Meaning", ACM Transactions on Database Systems, Vol. 4, No. 4, December 1979, pp. 397–434.

[Curry *et al* 82]

G. Curry, L. Baer, D. Lipkie, B. Lee, "Traits: An Approach to Multiple-Inheritance Subclassing", *Proceedings of the ACM SIGOA Conference on Office Information Systems,* University of Pennsylvania, 1982, pp. 1–9.

[Dahl *et al* 70]

O-J, Dahl, B. Myhrhaug and K. Nygaard, "SIMULA Common Base Language", Norwegian Computing Centre S-22, Oslo, Norway, 1970.

[Dasgupta *et al* 88]

P. Dasgupta, R. LeBlanc Jr., W. Appelbe, "The Clouds Distributed Operating System", *Proceedings of the 8th International Conference on Distributed Computing Systems*, San Jose, June 1988.

[Dijkstra 75]

E. W. Dijkstra, "Comments at a Symposium", in Selected Writings in Computing: A Personal Perspective, Springer-Verlag, New York, 1982, pp. 161-164. Also presented at the IBM/Newcastle Seminar on "Computers and the Educated Individual", 8-12 September, 1975, The University of Newcastle upon Tyne, England.

[Dijkstra 82]

E. W. Dijkstra, "'Why is Software So Expensive?' An Explanation to the Hardware Designer", in Selected Writings in Computing: A Personal Perspective, Springer-Verlag, New York, 1982, pp. 338-348.

[Dixon and Shrivastava 87]

G. N. Dixon and S. K. Shrivastava, "Exploiting Type-Inheritance Facilities to Implement Recovery in Object Based Systems", *Proceedings of 6th Symposium on Reliability in Distributed Software and Database Systems*, Williamsburg, March 1987, pp. 107-114.

[Dixon *et al* 87]

G. N. Dixon, S. K. Shrivastava and G. D. Parrington, "Managing Persistent Objects in Arjuna: A System for Reliable Distributed Computing", *Proceedings of the Workshop on Persistent Object Systems*, Persistent Programming Research Report 44, Department of Computational Science, University of St. Andrews, August 1987.

[Dixon 88]

G. N. Dixon, "Object Management for Persistence and Recoverability", Ph.D Thesis, Technical Report TR/276, Computing Laboratory, University of Newcastle upon Tyne, December 1988.

[Dixon *et al* 89]

G. N. Dixon, G. D. Parrington, S. K. Shrivastava and S. M. Wheater, "The Treatment of Persistent Objects in Arjuna", *Proceedings of ECOOP '89*, University of Nottingham, July 1989, pp. 169–204. Also in The Computer Journal, Vol. 32, No. 4, pp. 323–332, April 1989.

[Eager *et al* 88]

D. L. Eager, E. D. Lazowska, and J. Zahorjan, "The Limited Performance Benefits of Migrating Active Processes for Load Sharing", ACM SigMetric '88, 1988, pp. 63–72.

[Eswaran *et al* 76]

K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, "The Notion of Consistency and Predicate Locks in a Database System", Communications of the ACM, Vol. 19, No. 11, November, 1976, pp. 624–633.

[Ezhilchelvan and Shrivastava 86]

P .D. Ezhilchelvan and S. K. Shrivastava, "A Characterisation of Faults in Systems", *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, CA, January 1986, pp. 215–222.

[Geihs *et al* 86]

K. Giehs, H. Eberle, B. Schöner, M. Seifert, "Distributed Object Sharing in Heterogeneous Environments", IBM European Networking Center Technical Report No. 43.8610, IBM Deutschland GMBH, 1986.

[Gibbons 87]

P. B. Gibbons, "A Stub Generator for Multi-language RPC in Heterogeneous Environments", IEEE Transactions on Software Engineering, Vol. SE–13, No. 1, January, 1987, pp. 77–87.

[Goldberg and Robson 83]

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

[Gorlen *et al* 89]

K. E. Gorlen, S. M. Orlow, P. S. Plexico, Data Abstraction and Object-Oriented Programming in C++, John Wiley and Sons, 1989.

[Gray *et al* 76]

J. N. Gray, R. A. Lorie, G. R. Putzolu and I. L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base", in Modelling in Data Base Management Systems, ed. G. M. Nijssen, North–Holland, 1976.

[Gray 78]

J. N. Gray, "Notes on Data Base Operating Systems", in Operating Systems An Advanced Course, Lecture Notes in Computing Science, Vol. 60, Springer–Verlag, 1978, pp. 393–481.

[Gray 80]

J. N. Gray, "A Transaction Model", IBM Research Report RJ2895, IBM Research Laboratory, San Jose, CA, August 1980.

[Gray *et al* 81]

J. N. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, I. Traiger, "The Recovery Manager of the System R Database Manager", ACM Computing Surveys, Vol 13, No. 2, June 1981, pp. 223–242.

[Habert *et al* 90]

S. Habert, L. Mosseri, and V. Abrossimov, "COOL: Kernel Support for Object–Oriented Environments", *Proceedings of the joint ECOOP / OOPSLA Conference*, Ottawa, October 1990.

[Haerder and Reuter 83]

Haerder, T. and Reuter, A., "Principles of Transaction–Oriented Database Recovery", ACM Computing Surveys, Vol. 15, No. 4, December 1983, pp. 287–317.

[Haerder and Rothermel 87a]

Haerder, T. and Rothermel, K., "Concepts for Transaction Recovery in Nested Transactions", Technical Report RJ 5534 (56433), IBM Almaden Research Center, San Jose, CA, March 1987.

[Haerder and Rothermel 87b]

Haerder, T. and Rothermel, K., "Concurrency Control Issues in Nested Transactions", Technical Report RJ 5803 (58533), IBM Almaden Research Center, San Jose, CA, August 1987.

[Hall *et al* 76]

P. A. V. Hall, J. Owlett and S. J. P Todd, "Relations and Entities", in Modeling in Data Base Management Systems, G. M. Nijssen, ed., North-Holland Publishing Co., New York, 1976.

[Harland and Beloff 87]

D. M. Harland and B. Beloff, "OBJEKT - a persistent object store with an integrated garbage collector", ACM SIGPLAN Notices, Vol. 22 , April 87, pp. 70–79.

[Harland 88]

D. M. Harland, REKURSIV - Object Oriented Computer Architecture, Ellis Horwood, 1988.

[Herlihy and Wing 87]

M. P. Herlihy and J. M. Wing, "Avalon: Language Support for Reliable Distributed Systems", *Digest of Papers FTCS-17: Seventeenth Annual International Symposium on Fault-Tolerant Computing*, Pittsburgh PA, July 1987, pp. 89–94.

[Hoare 1981]

C. A. R. Hoare, "The Emporer's Old Clothes", 1980 Turing Award Lecture, ACM Turing Award Lectures - The First Twenty Years 1966–1985, ACM Press Anthology Series, ACM Press, New York, 1987, pp. 143–161.

[Hollberg *et al* 90]

U. Hollberg, H. Eberle, K. Geihs, R. Heite, H. Schmutz, "An Object Oriented View of Distribution", Technical Report No. 43.9004, IBM European Networking Center, Heidelberg, July 1990.

[Horning *et al* 74]

J. J. Horning, H. C. Lauer, P. M. Melliar-Smith and B. Randell, "A Program Structure for Error Detection and Recovery", in Lecture Notes in Computer Science, Vol. 16, Springer-Verlag, Berlin, 1974, pp 171–187.

[ISO7498]

ISO/TC97/SC16/WG1, "Information Processing Systems -- Open Systems Interconnection -- Basic Reference Model", ISO Standard IS7498, 1984 (also CCITT recommendation X.200).

[ISO8073]

ISO/TC97/SC16/WG1, "Information Processing Systems -- Open Systems Interconnection -- Connection Oriented Transport Protocol Specification", ISO Standard IS8073, 1984.

[Jacobson 82]

D. M. Jacobson, "Transactions on Objects of Arbitrary Type", Technical Report 82–05–02, Department of Computer Science, University of Washington, Seattle, WA, May 1982.

[Jessop *et al* 82]

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

[Jones and Rashid 86]

M. B. Jones and R. F. Rashid, "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems", *Proceedings of OOPSLA '86*, September 1986, pp. 67–77.

[Jul *et al* 88]

E. Jul, H. Levy, N. Hutchinson, A. Black, "Fine-grained Mobility in the Emerald System", ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988, pp. 109–133.

[Kent 78]

W. Kent, Data and Reality, North-Holland Publishing Co. New York 1978.

[Khosafian and Copeland 86]

S. N. Khosafian and G. P. Copeland, "Object Identity", *Proceedings of OOPSLA '86*, September 1986, pp. 406–416 *also in* Readings in Object Oriented Database Systems, S. B. Zdonik and D. Maier (eds.), Morgan Kaufman, 1990, pp. 37–47.

[Knight and Leveson 86]

J. C. Knight and N. G. Leveson, "An Empirical Study of Failure Probabilities in Multi–Version Software", *Digest of Papers FTCS–16: Sixteenth Annual International Symposium on Fault–Tolerant Computing*, Wien, July 1986, pp. 165–170.

[Knuth 73]

D. Knuth, Sorting and Searching, Volume 3, The Art of Computer Programming, Addison–Wesley, Menlo Park CA, 1973.

[Kohler 81]

W. H. Kohler, "Survey of Techniques for Synchronization and Recovery", ACM Computing Surveys, Vol. 13, No. 2, June 1981, pp. 149–183.

[Kroeger *et al* 90]

R. Kroeger, M. Mock, R. Schumann, "The RelaX Architecture – Overview and Interfaces", COMANDOS Working Paper GMD–RelaX 004, February 1990.

[Lampson and Sturgis 79]

B. W. Lampson and H. E. Sturgis, "Crash Recovery in a Distributed Data Storage System", Unpublished internal report, Xerox PARC, April 1979.

[Laprie 89]

J. C. Laprie, "Dependability: A Unifying Concept for Reliable Computing and Fault Tolerance", in Dependability of Resilient Computers, T. Anderson (ed.), BSP Professional Books, Oxford, 1989, pp. 1–28.

[Lazowska *et al* 81]

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

[LeBlanc and Wilkes 85]

R. J. LeBlanc and C. T. Wilkes, "Systems Programming with Objects and Actions", *Proceedings of the 5th International Conference on Distributed Computing Systems*, May 1985, pp. 132–139.

[Lee and Anderson 90]

P. A. Lee and T. Anderson, Fault Tolerance: Principles and Practice, Second, Revised Edition, Springer–Verlag, 1990.

[Lee *et al* 80]

P. A. Lee, N. Ghani and K. Heron, "A Recovery Cache for the PDP–11", IEEE Transactions on Computers, Vol. C–29, No. 6, June 1980, pp. 546–549.

[Leiner *et al* 85]

P. J. Leiner, R. Cole, J. Postel, D. Mills, "The DARPA Internet Protocol Suite", IEEE Communications Magazine, Vol. 23, No. 3, March 1985, pp. 29–34.

[Lippman 89]

S. B. Lippman, C+ + Primer, Addison–Wesley, 1989.

[Liskov *et al* 79]

B. Liskov, R. Atkinson, T. Bloom, J. E. B. Moss, C. Schaffert, R. Scheifler and A. Snyder, "Clu Reference Manual", Technical Report MIT/LCS/TR–225, MIT Laboratory for Computer Science, Cambridge, Mass., October 1979.

[Liskov and Scheifler 83]

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

[Liskov 84]

B. Liskov, "Overview of the Argus Language and System", Programming Methodology Group Memo 40, Massachusetts Institute of Technology Laboratory for Computer Science, February, 1984.

[Liskov 88]

B. Liskov, "Distributed Programming in Argus", Communications of the ACM, Vol. 31, No. 3, March 1988, pp. 300–312.

[Lomet 77]

D. B. Lomet, "Process Structure, Synchronisation and Recovery using Atomic Actions", *Proceedings of ACM Conference on Language Design for Reliable Software*, SIGPLAN Notices, Vol. 12, No. 3, March 1977, pp. 128–137.

[Marques and Guedes 89]

J. A. Marques and P. Guedes, "Extending the Operating System to Support an Object-Oriented Environment", *Proceedings of OOPSLA '89*, October 1989, pp. 113-122.

[Melliar-Smith and Randell 77]

P .M. Melliar-Smith and B. Randell, "Software Reliability: The Role of Programmed Exception Handling", ACM SIGPLAN Notices, Vol. 12, No. 3, 1987, pp. 95-100.

[Merlin and Randell 78]

P. M. Merlin and B. Randell, "State Restoration in Distributed Systems", *Digest of Papers FTCS-8: Eighth Annual International Symposium on Fault-Tolerant Computing*, Toulouse, June 1978, pp. 129-134.

[Meyer 88]

B. Meyer, Object-oriented Software Construction, Prentice-Hall, 1988.

[Mohan *et al* 86]

C. Mohan, B. Lindsay, and R. Obermarck, "Transaction Management in the R* Distributed Database Management System", ACM Transactions on Database Systems, Vol. 11, No. 4, December 1986, pp. 378-396.

[Mohan *et al* 89]

C. Mohan, D. Haderle, B. Lindsay, B. H. Pirahesh, P. Schwartz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", Research Report RJ 6649R, Data Base Technology Institute, IBM Almaden Research Center, September 1989.

[Moon 86]

D. Moon, "Object-Oriented Programming with Flavors", *Proceedings of OOPSLA '86*, September 1986, pp. 1-8.

[Moore 82]

J. D. Moore, "Simple Nested Transactions in LOCUS: A Distributed Operating System", M.Sc. Dissertation, University of California at Los Angeles, 1982.

[Morrison *et al* 89]

R. Morrison, F. Brown, R. Connor, A. Dearle, "The Napier88 Reference Manual", Technical Report PPRR-77-89, Universities of Glasgow and St. Andrews, 1989.

[Moss 81]

J. E. B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing", Technical Report MIT/LCS/TR-260, Massachusetts Institute of Technology, Laboratory for Computer Science, April, 1981.

[Moss 87]

E. Moss, "Log Based Recovery for Nested Transactions", *Proceedings of the 13th International Conference on Very Large Data Bases*, Brighton, September 1987.

[Nelson 81]

B. J. Nelson, "Remote Procedure Call", Ph.D. Thesis, Technical Report CMU-CS-81-119, Department of Computer Science, Carnegie-Mellon University, 1981.

[Nett *et al* 85]

E. Nett, K. E. Grosspietsch, A. Jungblut, J. Kaiser, R. Kroger, W. Lux, M. Speicher, H. W. Winnebeck, "Profemo: Design and Implementation of a Fault Tolerant Distributed System Architecture", GMD-Studien, No. 100, Technical Report, GMD, St. Augustin, June, 1985.

[O'Brien *et al* 86]

P. O'Brien, B. Bullis and C. Schaffert, "Persistent and Shared Objects in Trellis/Owl", *Proceedings of the International Workshop on Object-Oriented Databases*, Pacific Grove, CA, September 1986.

[Oki 83]

B. Oki, "Reliable Object Storage to Support Atomic Actions", Technical Report MIT/LCS/TR-308, Massachusetts Institute of Technology, Laboratory for Computer Science, May, 1983.

[Oppen and Dalal 81]

D. C. Oppen and V. K. Dalal, "The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment", Technical Report OPD–T8103, Xerox Office Products Division, Systems Development Department, October 1981.

[Parrington 88]

G. D. Parrington, "Management of Concurrency in a Reliable Object–Oriented Computing System", Ph.D Thesis, Technical Report TR/277, Computing Laboratory, University of Newcastle upon Tyne, December 1988.

[Parrington and Shrivastava 88]

G. D. Parrington and S. K. Shrivastava, "Implementing Concurrency Control in Reliable Distributed Object–Oriented Systems", *Proceedings of ECOOP '88*, Norway, August 1988.

[Parrington 90]

G. D. Parrington, "Reliable Distributed Programming in C++: The Arjuna Approach", *Proceedings of the USENIX 2nd C++ Conference*, San Francisco, April 1990, pp. 37–50.

[Pitts 88]

D. Pitts, "Recovery in the *Clouds* Kernel", *Proceedings of the 7th International Conference on Reliable Distributed Systems*, Columbus, October 1988, pp. 167–176.

[Polya 57]

G. Polya, How to Solve It, Second Edition, Princeton University Press, 1957, p. 173.

[Postel 81a]

J. B. Postel, "Internet Protocol", DARPA Internet Program Protocol Specification, September 1981.

[Postel 81b]

J. B. Postel, "Transmission Control Protocol", DARPA Internet Program Protocol Specification, September 1981.

[Powell 91]

D. Powell, "Fault Assumptions and Assumption Coverage", *Proceedings of the 2nd Open PDCS Workshop*, Newcastle upon Tyne, May 1991, Vol. 1, Ch. 5, I.

[Pu *et al* 88]

C. Pu, G. Kaiser, and N. Hutchinson, "Split–Transactions for Open–Ended Activities", *Proceedings of the 14th International Conference on Very Large Data Bases*, Los Angeles, California, September 1988, pp. 26–37.

[Raj *et al* 91]

R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, E. Jul, "Emerald: A General-Purpose Programming Language", Software – Practice and Experience, Vol 21, No. 1, January 1991, pp. 91–118.

[Randell 75]

B. Randell, "System Structure for Software Fault Tolerance", IEEE Transactions on Software Engineering, Vol. SE–1, No. 2, June 1975, pp. 220–232.

[Richardson and Carey 88]

J. E. Richardson, M. J. Carey, "Persistence in the E language", Software – Practice and Experience, Vol. 19, No. 2, December 1988, pp. 1115–1150,

[Rothermel and Mohan 89]

K. Rothermel and C. Mohan, "ARIES/NT: A Recovery Method Based on Write–Ahead Logging for Nested Transactions", *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam, August 1989, pp. 338–346.

[Rothermel and Pappe 89]

K. Rothermel and S. Pappe, "Open Commit Protocols for the Tree of Processes Model", IBM European Networking Center Technical Report No. 43.8909, IBM Deutschland GMBH, 1989.

[Rothermel and Pappe 90]

K. Rothermel and S. Pappe, "Open Commit Protocols Tolerating Commission Failures", IBM European Networking Center Technical Report No. 43.9006, IBM Deutschland GMBH, 1990.

[Rozier *et al* 88]

M. Rozier, V. Abrsossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, P. Léonard, S. Langlois, W. Neuhauser, "Chorus Distributed Operating Systems", Computing Systems, Vol 1, No. 4, 1988, pp. 305–370.

[Schaffert *et al* 86]

C. Schaffert, T. Cooper, B. Bullis, M. Kilian and C. Wilpolt, "An Introduction to Trellis/Owl", *Proceedings of OOPSLA '86*, September 1986, pp. 9–16.

[Schantz *et al* 86]

R. E. Schantz, R. H. Thomas, G. Bono, "The Architecture of the Cronus Distributed Operating System", *Proceedings of the 6th International Conference on Distributed Computer Systems*, IEEE Computer Society, May 1986.

[Scheifler and Gettys 86]

R. Scheifler and J. Gettys, "The X Windows System", ACM Transactions on Graphics, Vol. 5, No. 2, April 1986, pp. 79–109.

[Schumann *et al* 89]

R. Schumann, R. Kröger, M. Mock, E. Nett, "Recovery Management in the RelaX Distributed Transaction Layer", *Proceedings of the 8th Symposium on Reliable Distributed Systems*, Seattle WA, October 1989, pp. 21–28.

[Schwarz and Spector 84]

P. M. Schwarz and A. Z. Spector, "Synchronizing Shared Abstract Types", ACM Transactions on Computer Systems, Vol. 2, No. 3, August 1984, pp. 223–250.

[Sha 85]

L. Sha, "Modular concurrency control and failure recovery – Consistency, correctness and optimality", Ph.D. dissertation, Department of Electrical and Computer Engineering, Carnegie–Mellon University, 1985.

[Sha *et al* 88]

L. Sha, J. P. Lehoczky and E.D. Jensen, "Modular Concurrency Control and Failure Recovery", IEEE Transactions on Computers, Vol. 37, No. 2, February 1988, pp. 146–159.

[Shapiro *et al* 89]

M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Vaolt, "SOS: An Object Oriented Operating System – Assessment and Perspectives", Computing Systems, Vol. 2, No. 4, Fall 1989, pp. 287–338.

[Shrivastava 82]

S. K. Shrivastava, "A Dependency, Commitment and Recovery Model for Atomic Actions", *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, University of Pittsburgh, Pittsburgh, PA, July 1982, pp. 112–119.

[Shrivastava *et al* 87]

S. K. Shrivastava, L. Mancini and B. Randell, "On the Duality of Fault Tolerant System Structures", in Experiences with Distributed Systems, Lecture Notes in Computer Science, Vol. 309, Springer–Verlag 1987, pp. 19–37.

[Shrivastava *et al* 88]

S. K. Shrivastava, G. N. Dixon, F. Hedayati, G. D. Parrington and S. M. Wheater, "A Technical Overview of Arjuna: A System for Reliable Distributed Computing", *Proceedings of UK IT '88 Conference*, July 1988, pp. 601–605.

[Shrivastava *et al* 90]

S. K. Shrivastava, P. Ezhilchelvan, and M. C. Little, "Understanding Component Failures and Replication in Distributed Systems, ISA Project Report UNT/TR1, May 1990.

[Shrivastava *et al* 91]

S. K. Shrivastava, G. N. Dixon, G. D. Parrington, "An Overview of the Arjuna Distributed Programming System", IEEE Software, Vol. 8, No. 1, 1991, pp. 66–73.

[Skeen 81]

D. Skeen, "Nonblocking Commit Protocols", *Proceedings of the ACM SIGMOD International Conference on Management of Data,* Ann Arbor, Michigan, ACM, New York, 1981, pp. 133–142.

[Sollins 79]

K. R. Sollins, "Copying Complex Structures in a Distributed System", M.Sc. and Technical Report MIT/LCS/TR-2219, Massachusetts Institute of Technology, Cambridge, MA, May 1979.

[Spector and Schwarz 83]

A. Z. Spector and P. M. Schwarz, "Transactions: A Construct for Reliable Distributed Computing", ACM Operating Systems Review, April 1983, pp. 18–34.

[Spector *et al* 85]

A. Z. Spector, J. Butcher, D. S. Daniels, D. Duchamp, J. L. Eppinger, C. E. Fineman, A. Heddaya, P. M. Schwarz, "Support for Distributed Transactions in the TABS Prototype", IEEE Transactions on Software Engineering, Vol. SE–11, No. 6, June 1985, pp. 520–530.

[Spector *et al* 87]

A. Z. Spector, D. Thompson, R. F. Pausch, J. L. Eppinger, D. Duchamp, R. Draves, D. S. Daniels, and J. J. Bloch, "Camelot: A Distributed Transaction Facility for Mach and the Internet - An Interim Report", Technical Report CMU–CS–86–129, Department of Computer Science, Carnegie–Mellon University, June 1987.

[Spector *et al* 88]

A. Z. Spector, R. F. Pausch, G. Bruell, "Camelot: A Flexible Distributed Transaction Processing System", *Proceedings of IEEE Compcon,* Spring 1988, San Francisco, March 1988.

[Spector 89]

A. Z. Spector, "Distributed Transaction Processing Facilities", in Distributed Systems, S. Mullender (ed.), ACM Press Frontier Series, Addison–Wesley, 1989, pp. 191–214.

[Steedman 90]

D. Steedman, ASN.1 The Tutorial and Reference, Technology Appraisals, The Camelot Press, Trowbridge, Wiltshire, UK, 1990.

[Stroustrup 86]

B. Stroustrup, The C+ + Programming Language, Addison Wesley, 1986.

[Sun 88]

"Network Programming", Sun Microsystems, Mountain View CA, Revision A, May 1988.

[Tanenbaum and Mullender 81]

A. S. Tanenbaum and S. J. Mullender, "An Overview of the Amoeba Distributed Operating System", ACM Operating Systems Review, Vol. 15, No. 3, July 1981, pp. 51–64.

[Traiger 82]

I. L. Traiger, "Virtual Memory Management for Database Systems", ACM Operating Systems Review, Vol 16, October 1982, pp. 24–48.

[van den Bos and Laffra 89]

J. van den Bos and C. Laffra, "PROCOL - A parallel object language with protocols", *Proceedings of OOPSLA '89*, New Orleans, October 1989, pp. 95–102.

[Walker *et al* 83]

B. J. Walker, G. J. Popek, R. English, C. Kline and G. Thiel, "The LOCUS Distributed Operating System", *Proceedings of the 9th ACM Symposium on Operating System Principles*, Bretton Woods, New Hampshire, October 1983, pp. 49–70.

[Walker 85]

B. J. Walker, The Locus Distributed System Architecture, MIT Press, 1985.

[Weihl 89a]

W. E. Weihl, "Remote Procedure Call", in Distributed Systems, S. Mullender (ed.), ACM Press Frontier Series, Addison–Wesley, 1989, pp. 65–86.

[Weihl 89b]

W. E. Weihl, "Using Transactions in Distributed Applications", in Distributed Systems, S. Mullender (ed.), ACM Press Frontier Series, Addison–Wesley, 1989, pp. 215–236.

[Weikum 91]

G. Weikum, "Principles and Realization Strategies of Multi-level Transaction Management", ACM Transactions on Database Systems, Vol. 16, No. 1, March 1991, pp. 132–180.

[Weinreb and Moon 81]

D. Weinreb and D. Moon, LISP Machine Manual, Third Edition, March 1981.

[Weiser *et al* 89]

M. Weiser, A. Demers, C. Hauser, "The Portable Common Runtime Approach to Interoperability", ACM Operating Systems Review, Vol. 23, No. 5, December 1989, pp. 114–122.

[Weizenbaum 1976]

J. Weizenbaum, Computer Power and Human Reason, W. H. Freeman and Co., San Francisco, 1976, page 277.

[Wright 1927]

F. L. Wright, "The Architect and the Machine", in *In the* Cause of Architecture, F. Gutheim, ed., Architectural Record, McGraw-Hill, New York, 1975, pp. 131–134.

[Zdonik and Maier 90]

S. B. Zdonik and D. Maier (eds.), Readings in Object-Oriented Database Systems, Morgan Kaufmann, San Mateo, 1990.

Index