A Cache Coherence Protocol
for Concurrency Control and Recovery
in Distributed Object-oriented Systems

by

Sung-Gi Min

Submitted to the University of London
for the Degree of
Doctor of Philosophy in Computer Science

Department of Computer Science
Queen Mary & Westfield College

December 1993
Acknowledgements

I would like to express my gratitude to my supervisor Mrs. Jean Dollimore for her guidance throughout the course of this project.

I would also like to thank Professor G. F. Coulouris for his helpful suggestions at the early stages of my work.

Special thanks also due to all my friends for providing me with a pleasant environment away from home.

I wish to thank my family whose emotional support even from so far away kept me going for all these years.
Abstract

This thesis proposes a cache coherence protocol and an object-space model for transactions on objects in a distributed object-oriented system. The object-space model is an enhanced version of the client-server model. It reduces the effects of local system failures and supports independent service environments which enhance expandability and flexibility of the system. The cache coherence protocol maintains the consistency of replicas of a service in caches in context of transactions. The protocol includes a concurrency control protocol and a recovery mechanism to support transactions.

All transactions are required to use the most recent versions of services. If transactions violate this rule, the protocol tests whether the violated transactions can be excepted from the requirement without causing inconsistent state in the services they used. If they fail this test, they are aborted.

Because sending coherence information is a by-product of the two-phase commitment protocol using multicast, the protocol avoids unnecessary communication with repositories of services to update replicas in caches. The abortion rate of transactions is reduced by updating replicas as soon as possible and by considering operation sequences of transactions. By updating replicas and validating the transactions in caches, concurrency control and cache coherence are well distributed; the workload of conventional servers is distributed into caches.

A recovery scheme and a deadlock detection algorithm are proposed for use with the cache coherence protocol and the object-space model. The latter reduces the number of deadlock detection messages by up to half of those generated by other recent algorithms. The proposed recovery scheme is designed to minimise the cost of updating replicas in caches as well as to recover from failures of services. The persistent version of a passive service is stored as a base version and a history file which consists of a set of version updating information. Replicas in caches update themselves with the cache coherence protocol except when they fail to receive any update information. The passive service supplies missing versions enabling replicas in caches to be kept up-to-date, reducing the cost of restarting transactions and communication.
# Table of Contents

Abstract .................................................................................................................. 2
Acknowledgement ................................................................................................... 3
List of Figures .......................................................................................................... 8

1. Introduction ........................................................................................................ 10
   1.1 Motivation : Tightly-coupled Vs Loosely-coupled Architecture ................. 11
   1.2 Assumptions ................................................................................................ 16
   1.3 Contributions of the thesis .......................................................................... 16
   1.4 Structure of Thesis ...................................................................................... 18

2. Distributed Object-oriented Systems and Concurrency Control ..................... 20
   2.1 Distributed Systems ................................................................................... 20
      2.1.1 Distributed Systems ........................................................................... 20
      2.1.2 The Useful Techniques for Distributed Systems ................................. 22
      2.1.3 Client-Server Model .......................................................................... 23
   2.2 Object-oriented Systems ............................................................................ 24
      2.2.1 Object-oriented Concept ................................................................... 24
      2.2.2 Object Mobility ................................................................................ 25
      2.2.3 Atomic Objects ................................................................................. 26
   2.3 Atomic Actions : Transactions .................................................................... 27
   2.4 Concurrency Control ................................................................................ 30
      2.4.1 A Basic Model .................................................................................. 30
      2.4.2 Pessimistic versus Optimistic Concurrency Control ......................... 33
      2.4.3 Basic Pessimistic Concurrency Control Protocols ............................ 34
         2.4.3.1 Two-phase Locking .................................................................. 34
         2.4.3.2 Timestamping ......................................................................... 35
         2.4.3.3 Hybrid Protocol ...................................................................... 35
      2.4.4 Optimistic Concurrency Control Protocol ........................................ 36
2.4.5 Multiversion Concurrency Control ........................................... 36
  2.4.5.1 Multiversion Two-phase Locking .................................. 38
  2.4.5.2 Multiversion Timestamping ......................................... 38
2.4.6 Local Atomicity .................................................................. 39
2.4.7 Concurrency Control in Distributed Systems and
   Object-based Systems ................................................................ 41
  2.4.7.1 Concurrency Control in Distributed
            Systems ...................................................................... 41
  2.4.7.2 Concurrency Control in Object-related
            Languages .................................................................... 43

2.5 Recovery ............................................................................. 44
  2.5.1 Storage ......................................................................... 45
  2.5.2 Recovery Mechanisms .................................................... 45
    2.5.2.1 Intentions Lists ....................................................... 46
    2.5.2.2 Logs ....................................................................... 47
    2.5.2.3 Shadows .................................................................. 47
    2.5.2.4 Differential File ....................................................... 48

2.6 Replication .......................................................................... 48
2.7 Deadlock Detection ............................................................. 49
  2.7.1 Centralised and Distributed Deadlock Detection
            Algorithm ................................................................... 51
  2.7.2 Distributed Deadlock Models ......................................... 52
    2.7.2.1 Resource Deadlock Model ....................................... 52
    2.7.2.2 Communication Deadlock Model ............................. 53
    2.7.2.3 A Comparison of the Two Models ............................ 53

2.8 Conclusion .......................................................................... 54

3. Object Space ......................................................................... 55
  3.1 Overview ........................................................................... 55
  3.2 The Object Space .............................................................. 57
    3.2.1 Passive Object Space .................................................. 59
    3.2.2 Active Shared Object Space ....................................... 60
    3.2.3 The Required Architecture for the Object Space
            Model ...................................................................... 61

3.3 The Naming Scheme for Shared Objects .............................. 64
  3.3.1 Overview ..................................................................... 64
  3.3.2 The Inter-space Object Identifier ................................. 66
  3.3.3 A User Friendly Active Object Identifier ....................... 67
  3.3.4 Specialised Active Object Spaces and Their
            Identifiers ................................................................. 68
3.3.5 A Passive Object Identifier in a PSOS..........................69
3.4 Object Grouping......................................................70
  3.4.1 Object Structure.................................................70
  3.4.2 Object Relationships .........................................71
  3.4.3 Logical Object Grouping ......................................73
  3.4.4 Physical Object Grouping ....................................75
3.5 Object Structure and its Transfer................................76
  3.5.1 Process and Object Transition States .........................76
  3.5.2 The Structure of Atomic Objects .............................79
  3.5.3 A Standard Form for Flattened Objects.......................81
  3.5.4 Crossing space boundaries .....................................84
  3.5.5 A Communication Protocol ....................................87
3.6 Summary......................................................................88
4. Cache Coherence Protocol.............................................90
  4.1 Replication and Consistency .....................................90
  4.2 Cache Coherence Protocol ........................................93
  4.3 Concurrency Control...............................................97
    4.3.1. Local concurrency control ..................................99
  4.4 Comparison with Concurrency Control Protocols ...............100
  4.5 Transactions and Version Control ...............................103
  4.6 The Commitment Algorithm ......................................104
    4.6.1 Two-phase Commitment Protocol ..............................104
    4.6.2 Concurrent Commitment ......................................105
    4.6.3 Multiservice Commitment ...................................108
  4.7 Advantages of the Cache coherence Protocol ..................112
5. Correctness of Cache Coherence Protocol .........................114
  5.1 Sharable and Exclusive State ....................................114
  5.2 The Correctness of the Cache Coherence Protocol .............116
    5.2.1 Basic Definitions ..........................................116
    5.2.2 Cache Coherence Protocol .................................118
    5.2.3 Cache Coherence Protocol With Local
          Concurrency Control ........................................124
5.3 The User-defined Concurrency Control Object ................................................. 127

6. Recovery .............................................................................................................. 129
   6.1 Atomic Actions and System Recovery ............................................................ 129
   6.2 Active Recovery ............................................................................................ 130
   6.3 Passive Recovery .......................................................................................... 131
   6.4 The Recovery Scheme .................................................................................. 134
   6.5 An Example of Recovery .............................................................................. 138

7. Deadlock Detection ............................................................................................ 143
   7.1 Relationship to Prior Work ......................................................................... 145
   7.2 The Structure of a Local Deadlock Detector ................................................ 149
   7.3 Probes : A Priority Scheme ......................................................................... 150
   7.4 Deadlock Detection and Resolution ............................................................. 151
   7.5 An Example of Deadlock Detection ............................................................. 156
   7.6 The Cost of Deadlock Detection and Resolution ........................................ 159

8. Conclusions ........................................................................................................ 160
   8.1 Summaries and Conclusions ....................................................................... 160
       8.1.1 The object-space model ....................................................................... 160
       8.1.2 The cache coherence protocol ................................................................ 161
           A comparison of our cache coherence protocol with other work ............... 162
       8.1.3 Recovery scheme .................................................................................. 163
       8.1.4 Distributed deadlock detection ............................................................ 164

   8.2 Future Work .................................................................................................. 165
       8.2.1 Evaluation and Case Study ................................................................... 165
       8.2.2 Distributed repositories ........................................................................ 165
       8.2.3 Modifying operations ............................................................................ 166
       8.2.4 Persistent caches .................................................................................. 166

References ............................................................................................................. 167
List of Figures

Figure 1.1 (a) A tightly-coupled architecture, (b) a loosely-coupled architecture, and (c) a loosely-coupled architecture seen from the view point of a tightly-coupled one

Figure 3.1 (a) Conventional server model and (b) our object space model
Figure 3.2 The Architecture of our system
Figure 3.3 Naming scope of object spaces
Figure 3.4 The structure of an inter-space object identifier
Figure 3.5 Three-level naming scheme in ASOS
Figure 3.6 An example connecting our system with another system
Figure 3.7 An example of a PID structure
Figure 3.8 A diagram of object relations
Figure 3.9 The states of a) a process and b) an object
Figure 3.10 (a) an atomic object structure for the active object model (b) an atomic object structure for the passive object model
Figure 3.11 An example of an object and its flattened form

Figure 4.1 Communication between replicas
Figure 4.2 An example of the relationship between cache coherence and local concurrency control protocol
Figure 4.3 Message flow for a committing transaction

Figure 5.1 The structure of conventional data items and a complex object
Figure 5.2 An example of a constructed directed graph for a history h

Figure 6.1 (a) the structure of differential file and shadow file (b) an example of the effect of a slow transaction
Figure 6.2 An example of a base version and history files

Figure 7.1 (a) A message flow (a) in Sihna’s and Natarajan’s [85-1] and Roesler and Burkhard[88]’s algorithms, and (b) in our algorithm
Figure 7.2 A Local Deadlock Detector’s Structure
Figure 7.3 Deadlock detection algorithm - main procedures
Figure 7.4 Deadlock detection algorithm - subsidiary procedures
Figure 7.5 (a) Transaction dependencies in ASOSs and (b) their representation in local deadlock detectors
To my parents
Chapter I

Introduction

Due to rapid development of computer hardware during the last two decades, personal workstations have become cheaper and more widespread. These computers are interconnected via networks. Such distributed computing environments have become commonplace. Distributed computing provides a convenient way to share resources. It offers better reliability, expandability and autonomy than that of a centralised system. One of the best-known methods to share resources in distributed systems is the client-server model. Users access services via computer networks. However, the operational cost is still much more expensive than that of local access. Transferring data via a computer network requires more complex operations such as marshalling and several layers of pre- and post-processing within communication protocols. The computer network can be congested. Furthermore, centralised servers can be overloaded during peak hours. As a result, local processes are bounded by the status and speed of the computer network they use and by the speed of centralised services.

A distributed system is naturally a multiprocessor environment. Most servers are accessed concurrently by several users' processes. The consistency of the system must be preserved in the presence of such sharing. A process may not complete its task during its lifetime because the system may fail and concurrent processes may interfere with one another. Transaction systems are used to prevent system inconsistency due to system failures and interferences between concurrent processes. Moreover, the need for consistency is a major obstacle to replicating data among different computers. Maintaining the same consistent state of replicated data in different computers is very expensive in terms of time and processing power. As a result, most concurrency control protocols do not consider data replication. Even though a few concurrency control
protocols permit data replication, consistency checking is still done by a central agency in many of these protocols.

Object-oriented concepts have been used as a model to increase data and/or code sharing and to enhance encapsulation since the late 70's. In object-oriented systems, objects represent entities, and objects are related to each other via grouping and sharing mechanisms. Distributed object-oriented systems promise a great degree of sharing. The sharing mechanism of object-oriented systems provides sharing within a service and between services and distributed systems provide sharing between users. However, maintaining consistency becomes more complex. Unlike ordinary data items, objects are encapsulated and are intensively related to each other as well as being structured entities. Consistency control must be planned more carefully to satisfy and exploit the characteristics of objects.

We propose an enhanced client-server model and a new cache coherence protocol for a distributed object-oriented system. The model provides a high degree of distribution of servers' workload and of availability by replication of objects into caches. The cache coherence protocol ensures the consistency of replicated objects in caches without a heavy overhead and in a highly distributed way and provides concurrency control in the context of transactions.

1.1 Motivation : Tightly-coupled Vs Loosely-coupled Architecture

Once the multiprocessing concept has been introduced in a computer system, resource sharing, especially the sharing of data, becomes common and data must be maintained in a consistent state to avoid undesirable results.
Figure 1.1 (a) A tightly-coupled architecture, (b) a loosely-coupled architecture, and (c) a loosely-coupled architecture seen from the viewpoint of a tightly-coupled one.
There are two kinds of multiprocessor system architectures: tightly-coupled and loosely-coupled system architectures. The former has a set of processors with caches which share central memories and their processors are connected via a system bus. The latter has a set of processors, each of which has its own cache and memory, and their processors are connected via a communication channel. An element of the set is called a node and a node can have either a single-processor architecture or a tightly-coupled multiprocessor architecture. Data sharing in the loosely-coupled architecture is more expensive than in the tightly-coupled architecture because data can be shared via the central memories in the tightly-coupled architecture.

In a tightly-coupled architecture, caches are used to reduce the memory access time [Smith 82]. Data are used by several processors concurrently and data in the central memory can be updated from time to time. When a datum in the cache is updated by a processor, the cache tries to update the datum in the central memory either immediately or sometime later. The former is called write-through and the latter delayed-writing. The write-through policy keeps data more up-to-date but causes more communication traffic. When a datum is changed, the change must be reflected in the caches in other processors as well as in the central memory. When a datum is changed in a cache, the cache notifies all other caches in the system. The corresponding data in other caches are invalidated. They mark the affected data in their cache and re-fetch updated data from central memory. This is called write-invalidate.

Distributed computer systems are a kind of loosely-coupled architecture. The communication channel is called a computer network. However, distributed systems can provide not only a central storage facility to share data more easily but also shared services - the central storage facility itself is a service. The client-server model is commonly used to provide such services. If we regard such central storage services as central memories of the tightly-coupled architecture and a memory in the node as a cache for a processor, a distributed system can be seen as a tightly-coupled architecture with a very slow and unreliable bus. This view is very convenient because programs and data of shared services are usually stored in file storage and active services in servers are maintained in their volatile storage which
can be seen as central shared caches. Central shared caches can be seen as conventional servers.

In Figure 1.1 (a), a typical tightly-coupled architecture is shown, and a loosely-coupled architecture is given in Figure 1.1 (b). In Figure 1.1 (c), the loosely-coupled architecture can be seen as a tightly-coupled architecture. A processor (or processors) in a node can be a processing element and its main and secondary memory can be regarded as a local cache. The node (or nodes) handling shared information can be seen as a central memory. The caching of the main and secondary memories of nodes are very important in distributed systems [Schroeder 85] [Nelson 88] [Levine 87] [Renesse 89] [Satyanarayanan 89]. The same problem of inconsistency occurs as in a tightly-coupled architecture. To update caches, a central storage server notifies caches [Nelson 88] [Satyanarayanan 89] or local caches check any changes in central storage server [Sun 86]. However, concurrent access and consistency problems restrict the usage of caches in distributed systems. Several systems [Nelson 88] [Levine 87] do not use caches for concurrently accessible data because they use conflict-based concurrency control protocols, which are more suitable for a centralised environment than a distributed one. Another serious problem in distributed systems is that the computer network is very slow and it is even unreliable unlike the bus in a tightly-coupled architecture. As a result, frequent access of central storage is undesirable. In addition, caches are simple and are lightly loaded but servers are complex and can be heavily loaded. The execution speed of client nodes is restricted by the performance of the servers. Therefore, server nodes require very powerful machines to lessen this problem.

Unlike ordinary data which have only states and uniform operations (e.g. read/write for database records), an object has its own operations as well as its state which is hidden from its users. An object can update its state partially and can communicate with other objects. In our architecture objects are stored in central storage in servers and cached in client nodes, and the central storage provides recoverability for shared objects. Considering the abilities of objects and by viewing distributed systems as a tightly-coupled architecture as mentioned above, we can introduce a variant of the write-update method into our distributed object-oriented system. In the write-update method, when a cache updates, it sends
updating messages to other caches which update their states without the need to re-fetch them from the central storage in which they are stored. The write-invalidate method is adopted by the distributed shared virtual memory (DSVM) [Li 1989] [Wu 1990]. Li also suggested the write-update method which is used in Orca [Bal 87].

There are several major differences between our model and Distributed shared memory such as synchronisation, granularity and the entity of coherence, the method of solving inconsistency, and system persistency. Some DSM systems use reliable broadcast to support write-update, whereas we use unreliable multicast.

We do not require a synchronous method to update out-of-date caches which is costly in a local area network. In other words, we maintain a weak consistency, in which the consistency between a replica of an object in a cache and an (original) object stored in a server must be maintained, but consistency between replicated objects in caches is not essential nor enforced.

Consistency in the central storage is maintained in the context of transactions using the principle behind optimistic concurrency control with backward validation. However, weak consistency between replicas is voluntarily achieved via multicast communication. The synchronisation of replicas is done in terms of transactions and uses the same principles as optimistic concurrency control with forward validation.

All changes made by a transaction are kept in caches until the transaction tries to commit. These changes are sent to the corresponding objects in central storage via a multicast, which includes replicated objects in other caches. But the delivery of the information to the other caches is not guaranteed. Out-of-order or lost update messages to other caches are tolerated. This strategy achieves weak consistency of the object replicas in caches with minimum communication overhead.

If transactions share a cache, the cache performs local concurrency control.
1.2 Assumptions

We consider distributed systems which consist of hundreds of autonomous computers connected by a local area network. Long-haul networks are not considered. We also exclude the consideration of network partitioning. Each computer may have multiple processors and computers may be heterogeneous. Each computer may fail individually and a failed computer does not generate erroneous data. The local area network is capable of delivering point-to-point, multicast and broadcast datagrams. Corrupted datagrams are removed by the network. The delivery is not required to be reliable. A datagram may be lost or duplicated. The sending and receiving orders of datagrams may be different, too. We use the terms computer, machine, node, and host synonymously.

A cache is an abstractive term rather than physical one. Although more than one cache could co-exist in a node, for simplicity, we assume each node has one cache.

1.3 Contributions of the thesis

In this thesis, we present a cache coherence protocol and an Object Space model to support transactions on distributed objects. In addition, we present a recovery scheme and a deadlock detection algorithm which are suitable for use with the model. A cache coherence protocol is designed for:

1. **Inexpensive cache replication**: services are replicated into several caches. The consistency of these replicas is achieved by unreliable multicast communication when transactions commit and by the ability of objects to modify their own states.

2. **Distributed consistency control**: consistency control functionality is distributed into caches and further into each object replica. Consistency control includes concurrency control and recovery of transactions. The update messages of the commitment protocol are used as consistency messages and concurrency control messages; concurrency control does not require extra communication.
3. **Flexible concurrency control schemes**: the cache coherence protocol acts as a global concurrency control protocol with an optimistic aspect. Moreover, the protocol can integrate with local conflict-based protocols such as two-phase locking and timestamping protocols. A service can choose a specific concurrency control protocol depending on its requirements, as its local concurrency control protocol.

4. **Improved performance for the conventional optimistic protocol**: by considering the order of operations and by supplying the most recent states of objects as soon as possible, our protocol reduces the aborting rate of transactions. Our cache coherence protocol makes more effective use of multiple versions than the conventional optimistic protocol. It can therefore achieve more concurrency than that of the conventional optimistic protocol.

The object space model enhances the client-server model in the following aspects:

1. **Reduced workload of each server**: several functionalities of conventional servers are distributed into caches and central storage services. Services are guaranteed recoverability by central storage services, and concurrency control and recovery for transactions are distributed into caches.

2. **Reduced communication traffic**: by placing object replicas in service replicas which are located near users, remote invocation messages are greatly reduced. Furthermore, grouping of objects reduces the number of messages required for replication of objects.

3. **Reduced effects of local system failures**: when a replica fails, the effect on the other replicas is minimal. Because replicated objects in caches are temporary copies of objects in a central storage service and caches maintain transactions, a node failure does not seriously affect other replicated objects in other caches.

4. **Increased flexibility/expandability**: because each cache may have separated/independent environment from other caches and a cache
can provide a special executional environment for a new service, introducing a new service becomes easier.

We suggest a recovery scheme suitable for supporting our model. The recovery scheme is based on differential files and shadows. Shadows are used to reduce the disadvantages of differential files. The recovery scheme is designed to achieve persistency and to reuse out-dated replicas in caches. The recovery scheme stores recovery data so that the same data can be used as version information for replicated objects in caches. With the version information, out-dated replicas of services can be updated in caches without re-loading all their replicated objects from central storage services. This improves the overall efficiency of the system. It also assists the commitment phase, including concurrent commitment, of transactions and prevents commitment deadlocks for concurrent committing transactions.

A deadlock detection algorithm proposed in this thesis can be used with any dependency relation which causes processes to block in such a way that cycles may arise such as conflict-based or semantic-based concurrency control protocols. The algorithm is based on the edge-chasing mechanism with a priority scheme. It reduces the number of deadlock detection messages by up to half of those generated by algorithms suggested by Sihna and Natarajan [85-1], and Roesler and Burkhard [88], and it also reduces time delay and storage requirement.

1.4 Structure of Thesis

In chapter 2 we discuss related topics that are required to understand our model and the proposed cache coherence protocol. The topics include how to improve a distributed system, the client-server model, atomic actions, and concurrency control protocols. Related distributed systems and concurrent programming languages are also examined. In addition, recovery, replication and deadlock detection methods are discussed.

The object space model will be explained in chapter 3. This chapter includes our object definition, the definition of object spaces and the architecture of object spaces. A comparison between the client-server model and object space model is provided. The rest of the chapter focuses
on details of the object space model. Object naming, object grouping, a
standard object representation and the communication method used in
our model are described.

In chapters 4 through 5 we depict the proposed cache coherence protocol.
In chapter 4, a description of the protocol is given. After a brief description
of replication and consistency, the entire protocol is described, followed by
a comparison of our concurrency control protocol with optimistic
concurrency control. Then the description of the two-phase commitment
protocol follows. In chapter 5, we argue the correctness of our protocol. We
construct a directed graph based on local histories on objects, and show its
serialisability.

A recovery scheme and a deadlock detection algorithm are described in
chapter 6 and 7, respectively. In chapter 6, the recovery scheme is
explained. We discuss how it co-operates with the cache coherence
protocol. The deadlock detection algorithm is explained in relation to
those of Sihna and Natarajan [85-1], and Roesler and Burkhard [88] on
which the new algorithm is based.

Finally, in chapter 8, we summarise and evaluate our work and discuss
further work.
Chapter II

Distributed Object-oriented Systems and Concurrency Control

In this chapter, we will examine relevant topics required to understand the rest of this thesis. Firstly, the general properties of conventional distributed systems are described. These include well-known techniques to improve performance of distributed systems. Secondly, the object-oriented concept is discussed. This section gives a general view of object-orientation and other topics related with the distributed environment. Concurrency control, recovery, and replication will then be discussed. These three topics have been extensively researched in the past decade. We review the best-known strategies in each category. Finally, deadlock detection methods are described.

2.1 Distributed Systems

In this section, we explain why distributed systems have emerged and what kinds of techniques are used to improve their performance.

2.1.1 Distributed Systems

Due to technological changes in the last couple of decades and development in interconnection and communication technology, distributed systems have become widely used. With dramatic reduction of price, and increased performance of computers, it is common to find many organisations with a great number of computers nowadays [Champine 80] [LeLann 81] [King 83] [Wibur 87] [Coulouris 88].

A distributed system consists of nodes and communication media. Each node has at least one processing unit and some kind of volatile storage. Some nodes contain non-volatile storage such as disks, tapes, etc., and
such storage can be shared by other nodes via communication media. Communication media allow individual nodes to communicate with each other. The components communicate via message passing. A distributed system can conceal the separation of components. This concealment is called transparency. Transparency enables the users of a distributed system to see the system as a whole rather than as a collection of systems.

A distributed system provides several advantages over a centralised system. First, it increases performance. In interactive systems such as graphical user interfaces, fast response time is very important. A distributed system provides better response than a centralised system because users get adequate response via a local workstation. Second, a distributed system provides local autonomy of organisation, yet allows remote access by other users. Third, distributed systems are more reliable than centralised ones. A single failure of a centralised system means the disabling of the whole system but in a distributed system, a single failure is localised, thus reducing the effect of the failure. Fourth, a distributed system has very good expandability. When an organisation builds a computer system, it provides a lower entry cost and the modularity of a distributed system can ensure further expandability. Heterogeneous machines can be connected easily with the existing system. Fifth, by localising data and multiplicity of resources, a distributed system increases availability. Finally, a distributed system provides resource sharing. When each computer is stand-alone, each needs several resources, such as printers, disks, etc., independently. A distributed system provides a mechanism for sharing such resources.

A distributed system has several disadvantages, too. First, a distributed system requires a high degree of security. Most distributed systems have open architectures to achieve expandability; clients can easily access data via the computer network. Therefore, a distributed system can be more easily intruded upon. Second, components of a distributed system depend on a network to communicate with others. Therefore, distributed systems depend on the performance and reliability of the network. When a network is overloaded, the performance of the system is greatly reduced.
Finally, complexity of implementation is increased. It requires complex controls such as transparency and communication protocols.

2.1.2 The Useful Techniques for Distributed Systems

There are several techniques to improve the performance of distributed systems [Mullender 89]. First of all, a cache provides many benefits in a distributed system. Recently, many distributed file systems, such as Andrew [Satyanarayanan 89], CFS [Schroeder 85], Apollo DOMAIN [Levine 87], Sprite [Nelson 88], Bullet [Renesse 89], explore the benefits of caching. Most importantly, a cache provides data locality. Although there has been a dramatic reduction of computer hardware prices, the communication cost is still relatively higher when users access data remotely. By avoiding remote accesses, a distributed system reduces the workload of servers and communication traffic in the network. In the Unix file system, it is claimed that a cache with a moderate size reduces about 50% of disk traffic and 90% with larger caches [Ousterhout 85] and server loading is reduced by a factor of 2-5 [Nelson 88]. In the Andrew [Satyanarayanan 89] and the Bullet [Renesse 89] file system, entire files are cached. About 70% of files in a Unix system require whole-file transfer, and, by caching entire files, only open and close operations require server contact. Consequently, a cache reduces server workload and network traffic. The drawbacks of such systems are that they require local disks for acceptable performance and files which are larger than the local disk cannot be accessed at all. A cache can be built in volatile memory or in non-volatile memory. For instance, Bullet and NFS cache in volatile memory.

Another important technique is increasing availability and reliability by replication. Data is stored redundantly and losing some replicas does not affect the consistency of the data. It increases the availability of the system. When a single copy of a service is not available due to system crash or network failure, other copies can be used. For immutable information, no information exchange is required between replicas when they are accessed. However, there is a consistency problem between replicas when they are updated. Herlihy[86] describes a quorum consensus method which is a generalised form of majority Consensus [Thomas 79] and weighted voting [Gifford 79]. The method is commonly used to solve the consistency
problem between replicas. This kind of replication replicates entire sets of data rather than partial replication which is used by caching. Total replication is more related with system robustness and availability and partial replication is more related with system performance such as response time and throughput.

2.1.3 Client-Server Model

One of main objectives of distributed systems is to provide sharing of resources. Such resources are used via servers which handle them. Servers require a kind of protected subsystem to enhance security, abstraction, and maintenance [Wibur 87]. The client-server model [Watson 81] [Wibur 87] [Coulouris 88] is proposed to satisfy such a requirement. Servers provide such a protected system environment. Clients use servers via message passing mechanisms, and servers can invoke other servers. Servers wait until they receive requests from clients or other servers, and then they execute the requested operations and return the results of operations to the requester. Therefore, servers are passive. There are two kinds of servers: sequential and concurrent servers. A sequential server processes one request at a time and a concurrent server creates several processes and each one processes a given request. Usually, a process is created for a request and destroyed after execution of the request.

To satisfy requirements of a protected subsystem, servers should have the following properties [Watson 81] [Wibur 87]:

1. Recoverability: a server maintains recoverable objects. After a system crash, it recover its objects.

2. Abstraction: each server has its own interface definition. Clients can access the server only via operations defined in the interface.

3. Synchronisation: when a server permits concurrent operations, it provides synchronisation to prevent inconsistent states of objects.

4. Protection: a server provides a very well secured environment to prevent unauthorised access.
2.2 Object-oriented Systems

In this section, the concepts and properties of object-oriented systems will be explained. It includes the general concept of objects and examines related topics: object migration and the robust object concept.

2.2.1 Object-oriented Concept

There are several descriptions of the term object-oriented [Wegner 86,87,89] [Snyder 86] [Halbert 87] [Stein 87] [Ungar 86] [Blair 89] [Tomlinson 89]. The term object-oriented defined by Wegner is widely accepted. An object-oriented system provides notions of object, class, and inheritance. An object is defined with a state and a set of operations which maintain the state of the object. The state of an object is protected, so users can access its state via operations provided by the object. Such a protection is called encapsulation. A system that supports objects is called object-based. A class is a template from which objects can be created. The term abstract data type is sometimes used instead of class. The main difference of the two terms is that abstract data type describes the interface of operations but not their implementation. By hiding the implementation from users, it increases the flexibility of the implementation. The set of operations defined in the class/abstract data type is called the behavioural specification of objects belonging to the same class/abstract data type. Therefore, objects belonging to the same class have the same behaviour. The set of operations can be used as the basis of access control and there are no conceptual constraints for using operations concurrently and serially. Inheritance [Goldberg 83] permits the inheritance of behaviour from other classes called superclasses. Inheritance from a single class is called single inheritance and inheritance from multiple ones is called multiple inheritance. Wegner's definition is too restrictive. It excludes many interesting systems/languages such as Self [Ungar 86], Emerald [Black 87] [Jul 88,89]. For instance, there are other forms of resource sharing, called delegation [Stein 87] or prototyping [Lieberman 86]. Delegation allows objects to delegate responsibility for performing operations or finding values to one or more designated 'ancestors'. Systems/languages using delegation have no separation of a class and an object created by the class. For example, Self does not provide class but object and delegation. A system is called class-
based object-oriented if it supports class and inheritance or classless object-oriented if it supports prototyping. In our work, we use the term object-oriented system to refer to both class and classless object-oriented systems. The class concept can be simulated by objects and delegation.

Object-Oriented = objects + classes + inheritance | objects + delegation

Wegner[88] describes inheritance and delegation as an incremental modification mechanism - an hierarchical/acyclic relationship between objects. Both these mechanisms support hierarchical sharing between objects.

The object-oriented paradigm provides several advantages. Objects can be reused. A user can use objects which are created by another user and well tested. This reusability increases the productivity of software development. Another advantage is expandability. The given system can be easily expandable by creating new objects using existing objects. For example, a queue object can be created as a subclass of a list object and can use all operations defined in the list object. In addition, objects are well tested and used by several users. Therefore using existing objects enhances the robustness of the system.

2.2.2 Object Mobility

Objects can move across machine boundaries to reduce remote accessing and to achieve load balance between nodes. To support object mobility, naming must be location independent and object representation must be machine independent. In addition, executable codes of operations of an object must be compatible between machines or suitable codes must be provided for each machine type.

A system supporting object mobility requires policy and mechanism. Policy decides which object to move to which machine and when. Mechanism provides the means to move objects.

Name independency is related to local transparency and inter-object reference. When an object moves from one machine, M1, to another machine, M2, all objects referencing the moved object in M1 must
reference the object in M2. Furthermore, they must be able to reference the
object as before. To solve the problem, two mechanisms have been
proposed: 1) global identifier [Black 87] [Jul 88,89] and 2) local references
with proxies [Bennet 87,90]. A proxy replaces an object and forwards
messages sent to the object. The latter method may create too many
proxies in the system due to frequent moving, and to forward messages it
requires additional communication traffic.

Another problem is that the object representation structure may vary
depending on machines. An object has its state and operations. Passing
this information between machines having different types requires
adjustment because they may have different representation for data
and/or code.

Apart from naming and object representation, moving an object which is
involved in an activity in a machine is very complex. Searching and
removing all the relevant information such as that in a system stack may
cause heavy overhead - such a dependency is called resident dependency
[Artsy 89]. According to Artsy [89], moving a 100 kilobytes process, a typical
size process, take about 7.5 seconds; moving active entities is not desirable.

Another problem related to object mobility is reliability. The moving
mechanism may fail when machines involved in the move fail or when
communication failure occurs. Therefore, a system must provide a
recovery mechanism for such cases.

2.2.3 Atomic Objects

Objects which provide concurrency control and recovery are called atomic
objects [Liskov 82] [Wegner 87]. In an object-oriented system, constructing
atomic objects is simple with the hierarchical sharing mechanism.
Functionalities for atomic objects can be inherited from other classes
which support these functionalities.

There are two different approaches to the provision of atomic objects -
centralised and decentralised. In the former, there is a robust object space
which provides concurrency control and recovery for objects in the space
by the space manager. In the latter, the object space does not provide such
facilities but the objects themselves do so. In our model, a hybrid approach is adopted: the concurrency control and the recovery from transaction failures are done by objects in caches and the recovery from system failures are provided by a robust object space.

With the object-oriented concept, atomic objects are used to construct persistent storage. Unlike processes, objects include interfaces and states, and these objects are recoverable and control consistency by themselves. Some distributed systems such as Avalon[Detlfs 88], Arjuna[Dixon 89] [Parrington 88,89], and Argus[Liskov 82,88] use atomic objects. For instance, atomic objects are derived from recoverable class in Avalon.

2.3 Atomic Actions: Transactions

The state of a system is consistent if it satisfies the constraints of the system. In most cases, these constraints are unknown. An individual operation on the system may cause inconsistent state. For instance, transferring an amount of money from one account to another can cause such a problem. After subtracting the amount of money from one account and before adding the sum to another’s account, the system is in a temporarily inconsistent state. To solve this problem, the concept of transaction has been proposed. Each operation performed by a transaction is not visible to other users/transactions. If all the operations succeed, then the effect of the transaction is visible. In a large database system, the possibility of accessing the same resource at the same time is low, so the concurrent execution of transactions is very desirable. However, this concurrent execution may cause inconsistent state because of intervention of other transactions. The inconsistency problem is handled by a concurrency controller.

The formal definition of a transaction follows [Lampson 81] [Schwarz 84]:

**Definition of Transaction.** A transaction \( t \) is a linearly ordered sequence \((O_1,O_2,...,O_n)\) of operations such that if \( S \) is a consistent state, then \( t(S) = O_n(...(O_1(S))...) \) is also a consistent state.
A transaction usually starts with the `start_transaction` statement. After executing the statement, the transaction controller can recognise the new transaction. After finishing all operations, the transaction executes a commit statement. If the effect of the transaction observes the consistency constraints, then the effect is accepted; otherwise the transaction is aborted. The abortion of a transaction can be decided by users or the system and a transaction may be restarted. The validation as to whether the effect of the transaction observes the consistency constraints may be initiated by the commit statement or before executing each operation. A transaction will be atomic in the presence of a crash if the changes made by the set of operations is atomic in the presence of crashes: either all changes made are done or none of them are.

Atomicity properties [Lampson 81] [Schwarz 84] of transactions are:

1. Failure atomicity: operations of a transaction will be executed either entirely or not at all.

2. Permanence: if a transaction completes successfully, the effect of its operations will never subsequently be lost. Sometimes, permanence is referred to as durability.

3. Serialisability: Transactions appear indivisible or uninterruptable with respect to other transactions which may be executing concurrently.

There is another property called avoiding cascading aborts [Schwarz 84]: when a transaction aborts, no other transactions will be forced to abort as a consequence. It is an important property for permanence.

Operations of a transaction can be executed in different machines and transactions can be nested. Both can increase parallelism and limit the effects of failures. The former is called a distributed transaction and the latter nested transaction. A distributed transaction [Spector 89] [Coulouris 88] can be executed at several computers in the network. When the transaction is finished, all sub-transactions of the committing transaction must be committed together. Moss[85] proposed a model for nested transactions. A transaction has sub-transactions and the sub-transactions
can be executed concurrently. The parent transaction can manage failures of sub-transactions. Sub-transactions are committed only if the top transaction completes successfully. Nested transactions allow concurrency within a transaction and are more robust in the presence of partial failures. When a sub-transaction fails then the parent transaction may use another sub-transaction to complete the task [Weihl 88].

To ensure atomicity and to allow participants the option of aborting, the commitment of transactions is done in two-phases.

In the first phase, a coordinator, a server, gets an agreement among servers on behalf of a transaction. If they fail to arrive at an agreement, allowing the transaction may commit without causing inconsistency, then the transaction is aborted. In the second phase, the coordinator makes sure that participants do update their state even if there is a failure during this phase. Any protocol following this principle is called two-phase commitment [Gray 79] [Lampson 81] [Weihl 88].

Transactions pass through four phases to satisfy transaction atomicity: the preparation phase, the execution phase, the validation phase and the writing phase. Depending on concurrency protocols, all phases may not be required. For instance locking and timestamping does not require a validation phase. During the preparation phase, a transaction notifies the system that it is going to start. The transaction is assigned a transaction identifier. Depending on the concurrency control protocol, the order of transactions can be determined using transactions' identifiers or using dedicated orderable identities such as timestamps. Then the transaction executes operations in the next phase, and concurrency control and recovery managers monitor and control the activity of the transaction. Some transactions may be aborted during this phase. After the execution phase, the transaction tries to commit. Then validation of operations of the transaction is carried out. If transactions fail this validation phase, they are usually aborted. Changes made by transactions that pass the validation phase are done during the writing phase.
2.4 Concurrency Control

In this section, we describe a basic model and a concept of concurrency control which is based on Schwarz[84] and Weihl[89]. After a brief description of some basic concurrency control protocols, several other topics related to concurrency control will be discussed.

2.4.1 A Basic Model

Operations executed by concurrent transactions can be modelled by pairs of a transaction identifier and an operation. An operation consists of an invocation and its results and it is terminated if the invoker receives the result of the invocation. A transaction is said to be completed by either an abort or a commit operation. If a transaction starts any operations but has not completed, then the transaction is active. Two operations are commutative if the effects of the two operations are identical irrespective of the order in which they are executed, i.e. $O_2(O_1(S)) = O_1(O_2(S))$ where $S$ is the consistent state of the system before the two operations are applied.

A history, also known as a log or a schedule, represents a sequence of operations performed by transactions. An element of a history consists of a <transaction, operation> pair. A history is well-formed if a) a transaction must wait until one invocation completes before invoking another operation, b) in the history, there is no transaction both committed and aborted, and c) a transaction cannot invoke any operation after it completes. Only well-formed histories are considered in formalisations. A history is serial if it represents operations of sequentially executed transactions in some order. Two histories, $h$ and $h'$, are equivalent, $h(S)=h'(S)$, if the effect of the history, $h$, is the same as that of $h'$. A history, which represents operations of several transactions executed concurrently, is serialisable if there exists an equivalent serial history.

There are two kind of histories: an abstract and an invocation history. An abstract history describes the order in which operations affect objects, regardless of any reordering that might have been done by their implementation. For example, consider an account and two operations, credit and debit for the account. Let us suppose that the implementations of credit and debit operations are 1) to store the change in a temporary
value and 2) the actual change of the account is done at the end of a transaction. There is an abstract history executed by three transactions, $T_1$ - $T_3$:

$$
T_1 : \text{credit}(\text{Acc}, 100) \\
T_2 : \text{credit}(\text{Acc}, 50) \\
T_1 : \text{debit}(\text{Acc}, 50) \\
T_3 : \text{credit}(\text{Acc}, 100).
$$

If transaction $T_2$ commits before $T_1$, then the resulting abstract history, which is reordered by its implementation, is

$$
T_2 : \text{credit}(\text{Acc}, 50) \\
T_1 : \text{credit}(\text{Acc}, 100) \\
T_1 : \text{debit}(\text{Acc}, 50) \\
T_3 : \text{credit}(\text{Acc}, 100).
$$

An invocation history describes the order in which operations are actually invoked - it can be reordered before being applied to the objects. The mapping between the two kinds of history is a many-to-one relationship. An abstract history may be implemented by several invocation histories, but each invocation history is mapped into exactly one abstract history [Schwarz 84].

Dependencies represent relations between operations executed by different transactions. A transaction has dependencies on other transactions if it executes an operation and the operation has dependencies on other operations executed by other transactions on an object. For example, read and write operations have dependency on a datum. The dependency relation can be used to detect operations which may cause inconsistent state of the system.

Schwarz[84] defines the formal dependency relation as follows:

**Definition of Dependency** : there exists a dependency relation between two transactions: $T_2$ depends on $T_1$, denoted as $T_1 \prec_D T_2$, if $T_1$ performs an operation on an object and the object is later operated on by $T_2$. 
If a transaction $T_j$ depends on the transaction $T_i$, then it implies that $T_i$ precedes $T_j$. The transitive closure of $\prec_D$ is represented by $\prec_D^*$. A history is orderable with respect to a set of dependency relations if and only if each of the relations has a transitive closure that is a partial order. Let $D_P$ be a set of partially ordered transitive-closure dependency relations. Orderability with respect to $D_P$ is equivalent to serialisability. The condition can be relaxed with the dependency relation of commutative operations; cyclic dependencies caused by commutative operations can be ignored. A history is complete if it contains operations executed by several transactions which must be committed at every object and excludes operations of aborted transactions. A history is atomic if it is a completed history and serialisable with respect to $D_P$. Transactions generating an atomic history are guaranteed to be atomic.

For example, a locking protocol divides operations into read and write categories. There is a dependency relation between read and write and between write and write. The dependency relation between read and read can be ignored because the two operations are commutative. In other words, two different transactions executing read and write operations respectively cause dependency; one of them is blocked to prevent this dependency relation. Dependencies can be defined more finely by exploring type information of an object [Schwarz 84] [Herlihy 87-2] [Ng 89]. With such finely defined dependencies, some dependencies can be ignored to obtain more concurrency. For instance, two credit operations, which are kinds of write operations and which are executed by two different transactions, for an account in the above example can be executed concurrently if we consider the semantic of the account. This is because these two operations are commutative semantically even if they conflict syntactically.

There are several serialisability categories. Conflict serialisability, view serialisability, and final-state serialisability. Final-state serialisability includes view serialisability, and view serialisability includes conflict serialisability. For example, if a complete history is conflict serialisable then it also view and final-state serialisable [Papadimitriou 86]. Each category of serialisability is related to the amount of concurrency permitted by a concurrency controller (scheduler) which belongs to one of
serialisability categories. For instance, conflict serialisability is related to conflict-based concurrency controls such as two-phase locking and timestamping. View serialisability is related to multiversion concurrency control such as multiversion locking and multiversion timestamping.

2.4.2 Pessimistic versus Optimistic Concurrency Control

There are two commonly-used strategies for controlling the concurrency problem: pessimistic and optimistic. Pessimistic control assumes that any operation which causes a new dependency relation between transactions may cause an inconsistent state of the system. So it requests a permission to execute the operation. Requesting the permission initiates a validation procedure. If the request is rejected then the transaction that requested the permission is aborted or it waits until the permission is granted. Meanwhile optimistic control assumes that even though an operation may cause a cyclic dependency relation, the operation is permitted. It assumes that any sets of operations do not interfere with each other. After a transaction finishes all operations, the transaction requests its validation - the validation of the assumption. If any of them would cause an inconsistent state, then the transaction apologises and aborts the effect [Herlihy 90-1].

There are two distinct validation methods: backward and forward validation Hálder[84]. Backward validation ensures that the committing transaction has not been invalidated by the recent commitment of another transaction, whilst forward validation ensures that a committing transaction cannot invalidate any active transaction.

Both validation methods can be based on conflict or state. Conflict-based validation depends on pre-defined conflicts between pairs of operations, while state-based validation uses states of objects to validate transactions.

Pessimistic concurrency control protocols are based on conflict-based forward validation. Optimistic protocols can be used in both kind of validation methods. Forward validation requires less validation time because the required set of data/operations is much less than that of backward validation [Härder 84] [Herlihy 87-1]. This approach is suitable for centralised validation. In a distributed (replicated) environment,
forward validation has a disadvantage: the committing transaction must find out all related transactions with it and the affected replicas must be updated. In our concurrency control protocol, objects used by the committing transaction do the work on behalf of the committing transaction.

2.4.3 Basic Pessimistic Concurrency Control Protocols

In this section, three pessimistic concurrency protocols are discussed: standard two-phase locking, timestamping, and a hybrid protocol.

2.4.3.1 Two-phase Locking

Two-phase locking (2PL) was proposed by Eswaran[76]. A lock is created on a datum by a transaction which accesses the datum at the time of the first access. Until the lock is removed, other transactions are blocked. Locks are released altogether when the holding transaction is committed or aborted. Therefore, there are two phases: the growing and the shrinking phase. In the growing phase, a transaction requests locks for data items. In this phase, no lock is released. In the shrinking phase, the locks are released and no future lock is granted for the transaction. The main problem of the 2PL is that it can cause deadlock in which no transaction can progress. The order of the transaction relative to other transactions is determined when a transaction is blocked by a lock; the holding transaction precedes the blocked transactions. Therefore, the order is unpredictable; this property is called dynamic atomicity.

In two-phase locking, there are two kinds of locks: the read lock and the write lock. The read lock is a kind of shared lock and the write lock is exclusive. To increase concurrency, varieties of locks are introduced [Korth 83]. For example, read, intention-to-write lock, and commit lock. Transactions can set an intention-to-write lock over read lock. There are non-two-phase locking protocols [Mohan 85]. Locks can be released even before the holding transaction is committed, but after releasing a lock on a datum, the transaction cannot access the datum afterward.

Locking is a very good protocol to control hot spot data, and its performance is as good as optimistic if conflicts and deadlocks are unlikely.
Locking is also good for handling non-existing data, called phantoms [Härder 84]. However, locking has the following disadvantage: a) lock maintenance and deadlock detection causes a substantial overhead, b) there are no general purpose deadlock-free locking protocols providing a high degree of concurrency, c) two-phase locking requires locks to be kept until the end of a transaction [Eswaran 76], and d) a locking approach is too restrictive in many cases; locking may be necessary in the worst case [Kung and Robinson 81].

2.4.3.2 Timestamping

In timestamping [Berstein 81] [Reed 83], each transaction is assigned a timestamp and their operations are validated according to the types of operations and their timestamps. If any transaction has an operation that fails to pass the validation, it is aborted immediately and restarted. Timestamps are used to define the ordering of transactions. Because timestamps are totally ordered, no cyclic waiting occurs. A write operation is carried out if the data item was last read and written by older transactions, and a read operation if a datum was last written by an older transaction. In this scheme, the order of the transactions is determined when they start, i.e. before a conflict occurs. This property is called static atomicity.

Timestamping prevents deadlocks but transactions can be aborted unnecessarily [Reed 83]; it prevents deadlock with reduced concurrency. Timestamping can be seen as locking with pre-defined ordering.

2.4.3.3 Hybrid Protocol

In a hybrid protocol [Farrag 87] [Weihl 87] [Wang 88], transactions are assigned timestamps as in timestamping. Any read-only transaction has a lower timestamp than any update transaction; this ordering has the static atomicity. Between update transactions, the order is determined by invoking operations and obtaining locks; this has dynamic atomicity. Therefore, the protocol combines the properties of dynamic and static atomicity. The static property is used for read-only transactions and the dynamic property is used for update transactions. This kind of protocol is called hybrid [Weihl 89].
Conflict-based protocols are related to dependency as follows. In the 2PL, dependencies are defined between read and write operations. To prevent cyclic dependency it blocks operations which may cause cyclic dependencies. Due to blocking, transactions may cause cyclic waiting: deadlock. In the timestamp protocol, dependencies are also determined not only by operations but also by timestamps. Timestamps are used to decide whether the conflicting transaction must be blocked or aborted to prevent cyclic dependency and cyclic waiting. Both protocols ignore dependencies caused by commutative operations. In the hybrid protocol, the dependency relation between read-only and update transactions is determined by the timestamp order; read-only transactions have lower timestamps than update transactions running concurrently.

2.4.4 Optimistic Concurrency Control Protocol

The basic idea of the optimistic scheme is to defer the effect of dependency relations between read and write operations until a transaction tries to complete its operation [Kung 81]. To avoid dependencies between read and write operations, each write operation makes a temporary copy of the original state. The copy is validated when the transaction which generates this copy completes. If the copy does not pass the validation then the transaction is aborted. Validation between two transactions, $T_i$ and $T_j$, that run concurrently is done by the following rules: a) $T_i$ ($T_j$) does not read data items modified by a transaction $T_j$ ($T_i$) and b) $T_i$ ($T_j$) does not overwrite data written by a transaction $T_j$ ($T_i$).

This kind of concurrency control assumes that the domain of shared data is very large, so that the probability of a dependency relation which can cause an inconsistent state of the system is little. When most transactions are read-only, this scheme is very useful. This protocol may cause transaction starvation: a transaction is aborted repeatedly, so it can never be committed.

2.4.5 Multiversion Concurrency Control

The previous concurrency control protocols, except the optimistic one which may use multiple copies, have a common assumption: after a write operation on an object, all other transactions must read the state of the
object from the modified version unless the transaction that executed the write operation is aborted afterwards. Even in the optimistic protocol, read operations cannot see the overwritten state of an object; they can access the version with which their transactions started. In multiversion concurrency control [Papadimitriou 84,86], each write operation creates a new version of an object and old and new versions are kept, and read operations can choose any version of an object. By giving this freedom, more concurrency can be obtainable. There are several varieties of the multiversion concurrency control protocol depending on how to assign a version to read operations.

The dependency between operations executed by the transactions is that a transaction, $T_1$, has read operations on a version of data which is modified by another transaction, $T_2$. Let $h$ be a function that translates read operations into version read operations. A history is a multiversion history (MVH) if 1) the history contains a union of histories of each transaction and the union is translated by the function $h$, 2) if there is a dependency between two operations in a transaction history, then the same relation must exist in the MVH, and 3) before read operations there must be a transaction which generates a modified version of the data. A serial MVH is one-copy serial if for transactions $T_i$ and $T_j$ and data $X$, $T_i$ depends on $T_j$ implies that $T_j$ must be the last transaction preceding $T_i$ that writes into any version of $X$. In other words, there is no cyclic dependency between transactions in a dependency graph. A MVH is one-copy serialisable if it is equivalent to a one-copy serial MVH [Bernstein 83].

Conflict multiversion serialisability is used to determine whether a history is serialisable, based on conflict dependencies between operations. In conflict multiversion serialisability, a write operation followed by another write operation and a write operation followed by a read operation are not in conflict. In both cases the two operations are commutative because reversed orders are serialisable. The former is serialisable because both versions created by two write operations can survive. The latter is serialisable because one of two original histories can simulate the reversed history by reading the previous state of an object instead of the state created by the write operation. However, a read operation followed by a write operation is in conflict because the original
history cannot simulate two histories which are generated by reversing two operations.

2.4.5.1 Multiversion Two-phase Locking

The standard two-phase locking (2PL) can be extended into multiversion two-phase locking [Chan 82,85]. As in 2PL, transactions acquire a read lock on an object immediately before read operations. However, there is no write lock because write operations create versions of the object rather than conflicts as in 2PL. Instead of write lock, there is version lock in multiversion two-phase locking. Whenever a new version is created a version lock is set on the version. Before a transaction commits, it has to obtain commit locks for all objects read from the current version which is created by the latest committed transaction and from uncommitted versions which are created by active transactions. Read locks and committed locks on the same version can co-exist. However, read and commit locks and version and commit locks cannot co-exist. This means that a transaction cannot commit until 1) all transactions which have read locks on the current version of an object on which the committing transaction has its own version and 2) all transactions which have uncommitted versions from which the committing transaction has read are committed. All locks obtained by a transaction are kept until the transaction commits and a transaction is blocked if a required lock is not obtainable.

Transactions in multiversion two-phase locking can cause deadlock due to cyclic waiting. A transaction can be aborted by an abortion of another transaction if the transaction reads from the uncommitted version of an object which was created by the aborted transaction.

2.4.5.2 Multiversion Timestamping

Multiversion timestamping is an extended version of the standard timestamping. Therefore, each transaction has its timestamp assigned when it starts. Because there are several versions of an object, each version has its read and write timestamps in multiversion timestamping.
In multiversion timestamping [Bernstein 83] [Herlihy 87-2], a transaction can read a state of an object among several versions of the object. The transaction chooses a version as its appropriate version of the object. The appropriate version of an object for a transaction is the version whose write timestamp is smaller than that of the transaction and as large as possible. The final version of an object must be specified; it is the version whose write timestamp is the largest among its versions. A read operation of a transaction is always executed. It reads the state of an object from its appropriate version for the transaction. A write operation can proceed if the timestamp of the transaction is larger than the read timestamp of the appropriate version for the transaction. The condition for write operations is required to maintain the order of transactions in the ascending order of their timestamps. In other words, it prevents cyclic waiting among transactions: no deadlock occurs.

2.4.6 Local Atomicity

Even though each object involved in the operations of a transaction, maintains its own consistent state, the state of the system may not be consistent. As described earlier (see section 2.2.3), atomic objects provide such a consistent state. The problem faced by atomic objects is that they have only local information. The local atomicity ensures that at least one serialisation order of the committed transactions can be satisfied by the atomic objects. Weihl [89] describes several local constraints on individual objects that suffice to ensure the total atomicity of transactions using atomic objects. In other words, if all objects in a system observe one local atomicity constraint, then the state of the system is always consistent. Weihl[89] calls such constraints local atomicity properties. All definitions in this section are taken from [Weihl 89].

The formal definition of the local atomicity property is as follows:

**Definition of Local Atomicity Property**: A local atomicity property is a property, P, of specifications of objects such that if the specifications of every object in a system satisfies P, then every history in the system's behaviour is atomic.
There are three types of properties satisfying the above definition: dynamic, static, and hybrid properties. The informal property of protocols characterised by dynamic atomicity is: if a committed transaction conflicts with another committed transaction in a history, then some operations executed by one of the transactions must occur after the other transaction has committed. It solves conflicts by blocking these operations. In contrast, transactions in a history are serialisable in a predetermined order in static atomicity. Conflicts are solved by delaying and aborting transactions which cannot satisfy the predetermined order during the execution. In hybrid atomicity, transactions are divided into two groups, read-only and update transactions. Between read-only and update transactions, static atomicity is applied. Read-only transactions have older timestamps than any active update transactions; they precede any active update transactions. Between update transactions, dynamic atomicity is applied.

Let \( \text{permanent}(h) \) be a history excluding any operations of aborted transactions in \( h \). The formal definitions of the local atomicities are as follows:

**Definition of Dynamic Atomicity**: A relation on transactions, \( a \) and \( b \), is defined \( <a,b> \in \text{precedes}(h) \) where \( h \) is a history if and only if there exists an operation invoked by transaction \( b \) that terminates after transaction \( a \) committed in \( h \). A history \( h \) is Dynamic Atomic if \( \text{permanent}(h) \) is serialisable in every total order, preserving \( \text{precedes}(h) \) relations between transactions.

**Definition of Static Atomicity**: Let \( TS \) be a predetermined total order on transactions. A history \( h \) is Static Atomic with respect to \( TS \) if \( \text{permanent}(h) \) is serialisable in order to \( TS \).

**Definition of Hybrid Atomicity**: Let \( h \) be a history and \( TS \) be the partial order on transactions such that \( <a,b> \in TS(h) \) if both transactions \( a \) and \( b \) commit in \( h \) and if \( a \)'s timestamp is less than \( b \)'s timestamp. A history \( h \) is Hybrid Atomic if \( \text{permanent}(h) \) is serialisable in all total orders consistent with \( TS(h) \).
The basic concurrency control protocols have one of these local atomicities. For example, the standard two-phase locking is an example of dynamic atomicity, and the time-stamping protocol has static atomicity. Hybrid atomicity gets both the characteristics of dynamic and static atomicity. Weihl points out that the local atomicity property is not a protocol for synchronising transactions, but a semantic constraint on the amount of concurrency that can be permitted by an object.

2.4.7 Concurrency Control in Distributed Systems and Object-based Systems.

In this section, we will discuss concurrency control schemes in distributed systems and object-oriented systems. Many distributed systems support concurrency control schemes for transactions. Concurrency control schemes can be centralised or distributed, and they also control a single or multiple version of objects. Concurrent object-oriented languages support some aspects of concurrency control for synchronising the processes. We will also examine these systems to illustrate access-based systems and to discuss the object-oriented properties for distributed systems.

2.4.7.1 Concurrency Control in Distributed Systems

Conflict-based concurrency control schemes are the most widely used in distributed systems and operations for concurrency control is adopted several systems via the object-oriented concept, but objects are not involved in validation because conflict-based concurrency protocols guarantee validation of operations implicitly. We will examine several distributed systems which have good characteristics.

In the Argus system [Liskov 82,88] [Oki 85] there is a unit of concurrency control agency called a guardian. Guardians survive from system failures and recover their state from the stable storage. A guardian is a kind of big atomic object that includes system or user-defined atomic objects which are guaranteed atomicity by the guardian. To recover from system failures, a guardian maintains stable storage constructed by logging. Atomic objects provide a set of operations to access and manipulate them. An atomic object is a basic unit representing the state of the guardian. Synchronisation is done by means of locks, using the standard two phase
locking protocol, and operations are divided into read and write operations. Recovery of atomic objects is done by using versions. Before an operation modifies the base object, the base object is copied into volatile memory and the operation is done on the copy. Transactions executing operations on an object obtain locks on that object. If a transaction commits then the copy becomes the base version of the object and is saved into the stable storage. To commit a transaction, guardians which are visited by the transaction cooperate by using the two phase commitment protocol.

Arjuna [Dixon 89] [Parrington 88,89] defines super classes to control concurrency. In this system, a lock is treated as an object, whose class is maintained by LockManager. Facilities for atomic actions are provided by AtomicAction class. AbstractRecord class provides a facility to record information for recovery, concurrency and persistency. In Arjuna, obtaining locks is defined in the body of an operation; users defining the operation of an object should specify explicitly when it has to obtain locks.

Avalon [Detlefs 88] introduces recoverable, atomic, and subatomic classes to support synchronisation and recovery. The recoverable class handles recoverability. Recoverable operations are performed at virtual memory level and the effects are saved onto stable storage. The atomic class and subatomic class are subclasses of the recoverable class. The atomic class supports object atomicity as in the Argus system. Recovery is guaranteed by the recoverable class and synchronisation is done by using locking. Transaction consistency is guaranteed by the Avalon run-time system performing special abortion processing. One drawback of the atomic class is that locks have to be obtained by explicit locking operations. The subatomic class provides fine granularity concurrency. The subatomic class supports two level synchronisations: operation-level and transaction-level synchronisation. Operation-level synchronisation called short-term provides mutual exclusion for transactions like a conventional monitor, and transaction-level synchronisation called long-term guarantees transaction consistency by using commit and abort processing. The subatomic class provides more concurrency using semantic information of the type; with more knowledge of commutative operations, transactions execute more operations concurrently.
In the Emerald [Black 87] [Jul 88,89], concurrency between objects and within an object is supported. Some objects are active, each of which has its own process. Within the object, multiple processes can be executed. Concurrency control is supported by a traditional monitor. The monitor synchronises processes using conventional condition variables; Emerald only supports short-term synchronisation - no transaction facility. It doesn't provide recoverable objects in the system, nor does it support transactions.

Argus is a class-based system rather than object-oriented system. It supports dynamic generation of guardians and atomic object sharing operations, but it does not support a hierarchical sharing mechanism. Arjuna, Avalon, and Emerald are object-oriented systems. They support hierarchical sharing mechanisms. These systems provide a very well organised structure to support atomic actions, concurrency control, and recovery via the hierarchical sharing mechanism. Emerald is a distributed object-oriented language rather than a distributed object-oriented system. Emerald classes support neither inheritance nor prototype, but they use a composition method instead.

2.4.7.2 Concurrency Control in Object-related Languages

Several object-oriented languages such as Concurrent Smalltalk [Yokote 86,87], Hybrid [Nierstrasz 87] and Actor++ [Kafura 89] implement concurrency control as a part of the language. The concurrency control in this kind of language is operation-level concurrency control rather than transaction-level one.

ConcurrentSmalltalk applies a state-based approach using atomic objects. An atomic object, having properties similar to the monitor, serialises incoming messages. It also introduces secretary objects for conditional critical regions. This kind of concept of atomic objects only partly addresses requirements for concurrency control in distributed systems which support transaction facilities, but the concept of encapsulation is very useful to implement independent concurrency strategies.

Hybrid and Actor++ use Interface Control. Interface control is based on direct control of the object interface; it is also called access-based control.
All incoming messages to a shared object are usually queued, and then a message is executed when the selector (the operation) of the message can access its internal state. Hybrid uses *open* and *close* operations with delayed queues for synchronising, and each method in it explicitly contains these operations in statements. The *close* operation blocks the access to the method until it is opened by an *open* operation - the method becomes unavailable and all messages referring to the method are queued at its queue. A problem in this language occurs when a subclass defines a new method requiring its own delay queue which is not present in its superclass. There is no way to control the delayed queue in methods of its superclass. Hence, all superclass methods needing *open* and *close* have to be revised to refer to this delay queue in their definitions.

Actor++, using the Actor model defined by Agha [Agha 86], achieves synchronisation with a *become* operation and message passing. A *become* operation replaces its behaviour with a new behaviour that processes the next message. However, a method defined in a subclass may invalidate its superclass methods because the method in a superclass does not recognise new subclass’ method for the *become* operation [Kafura 89].

The problem in Hybrid can be solved by using an automatic controller for the open and close statement. The controller can detect whether an operation is available by testing conflict between operations. Conflicts can be determined by syntactic analysis. Access-based concurrency control gives another aspect to control concurrency over objects. In Actor++, an actor consists of a message queue and its behaviour. This structure can be used to construct a compound object to combine a concurrency controller and a normal object.

### 2.5 Recovery

In this section, recovery is examined. Recovery is used to restore the consistent state of a system after a failure. First, we will discuss types of storage and then well-known recovery mechanisms.
2.5.1 Storage

In distributed systems, there are three kinds of storage: volatile, non-volatile and stable storage. Volatile storage is vulnerable to system crashes and loss of power. Main memory and caches in computers are typical examples of volatile storage.

Non-volatile and stable storage [Lampson 81] are more robust than volatile storage: they do not lose their information by losing power supply. Updating information in non-volatile storage is not atomic. Information in non-volatile storage may be damaged by a crash during updating. Stable storage usually consists of pairs of pages of non-volatile storage and both usually have the same information. Unlike non-volatile storage, stable storage maintains atomic updating. Writing on the stable storage is done by two individual operations which execute sequentially. At recovery from a crash, each element of stable storage has one of the following: 1) the pair have the same information, 2) one has undamaged information and the other damaged, or 3) both have undamaged but different information. In the first case, the recovery manager does nothing. In the second case, the damaged copy is restored from the undamaged one. In the third case, by choosing one of two copies and replacing the other by the chosen copy, the state of the stable storage is restored.

2.5.2 Recovery Mechanisms

As mentioned earlier (see section 2.3), one of properties required for the transaction facility is failure-atomicity. This property is guaranteed by a recovery scheme. There are two kinds of recovery: recovery from transaction abortions and from system failures. Recovery from transaction failure nullifies the effect of aborted transactions.

Weihl[88] describes five types of failures for which recovery is supported: transaction failure, server failure, node failure, media failure, and communication failure. To summarise:

1. Transaction failure: this kind of failure can happen when a transaction is aborted. A transaction can be aborted by users or by systems. A system usually decides to abort transactions when they fail
to meet consistency constraints or when they cannot carry on their operations due to other failures in the system.

2. Server failure: when a server runs out of resources, when a software error occurs, or when the hardware on which it runs fails, a service can fail. On recovery from the failure, it must undo the effects of uncommitted transactions and restore the consistent state.

3. Node failure: node failure can be represented by combined failures of transactions and servers in the node.

4. Media failure: it occurs when non-volatile storage of a node fails. The consistent state of a system can be restored from a backup copy or from other redundant copies stored on other media.

5. Communication failure: it occurs when communications facilities are damaged. Servers may be unavailable until failures are fixed, and the system state can be inconsistent due to partitioning.

Reliability is defined as the probability at which failures occur during an given time interval. By reducing the probability, the given system becomes more reliable. Lampson [81] describes errors and disasters as undesirable events in a system. Errors can be corrected later but disasters cannot be repaired. Media failure is an example of disasters. A component of the system is stable if it works in the presence of any number of errors. A stable component may fail when a disaster happens.

There are two points of view from which to handle failures in systems. One is transaction-oriented recovery and the other system-oriented. Intentions list, shadow paging and write-ahead logging are the best known techniques of the first point of view. Recovery techniques for system failures are logs, shadows, and differential files. We will not describe shadow paging and write-ahead logging because of their similarity with shadows and logs, respectively.

2.5.2.1 Intentions Lists

An intentions list [Lampson 81] contains information describing the intended changes made by a transaction. At the end of a transaction’s
operations, an intentions list of the committing transaction is written out to stable storage. When the transaction is approved to commit, the affected objects update their states according to the intentions list and the intentions list is eventually removed. If the transaction fails to commit, the intention list is discarded.

On recovering from a crash, the system checks the intentions lists in the stable storage. If there are no such lists, the recovery is done. If incomplete intention lists are found, then the system discards them; transactions related with such intention lists are aborted. Whenever any complete intentions list is found, objects are updated.

2.5.2.2 Logs

The log-based approach [Gray 79] [Schwarz 84] relies on a redundant representation of the database on an append-only log. An operation updating a data object creates a log element. A log element usually contains the transaction identifier, the object identifier, and old and new values. Unlike intentions lists, it permits immediate updating of objects as soon as the log of an operation is made.

Before updating data values, old values are stored on stable storage and before committing a transaction, all log elements generated by the transaction are also saved. To undo effects of an uncommitted transaction, a recovery manager scans log elements of the transaction backward and restores old values.

2.5.2.3 Shadows

In shadow-based schemes [Lorie 77] [Gray 81], two copies of an object are kept while the transaction is still active: the modified copy and the original copy which is not affected by the transaction. The latter is called shadow. Whenever the transaction succeeds, the original copy is replaced with the modified copy; otherwise the modified copy is discarded.

Before committing a transaction, its modified copy (lists of shadow pages) with the pre-committed transaction list is stored on the stable storage. On re-starting from a system crash, a recovery manager examines the pre-committed list and the committed list. Whenever a transaction appears in
the pre-committed list but not in the committed list, effects of the transaction are re-applied.

2.5.2.4 Differential File

With the differential file scheme [Severance 76] [Aghili 82], each logical file comprises two categories: a read-only base file and read-write differential file. The base file remains unchanged until reorganisation. All updates are confined to the differential file. The differential file is divided into an *Add* file and a *Deletion* file. Each file $R$ is considered a view, $R = (Base \cup Add) - Deletion$ where $Base$ is the read-only base portion of $R$.

In the distributed differential file scheme [Aghili, 82], each transaction has been assigned a unique timestamp and the tuples in the *Add* and *Deletion* files have an extra field for such a timestamp. While a transaction is active, its updates go to its local add file, $A_I$, and delete file, $D_I$, files. When the transaction commits, $A_I$ and $D_I$ are appended to the global add file, $A_g$, and global delete file, $D_g$, and then are stored on the stable storage. The timestamp of the committed transaction is, then, written to a committed transaction list. If a transaction aborts, its files are discarded. To recover from system crashes, a view of a file $R$ is used that consists of the tuples whose timestamp fields are contained in the committed transaction list.

2.6 Replication

A replicated object is a data object whose state is stored redundantly at multiple locations to enhance availability and reliability; objects are accessible even when some of the copies are unavailable due to site failures, and a user can use a near by object [Thomas 79]. Herlihy[86]'s Quorum-Consensus replication method is a general version of Thomas[79]'s majority Consensus and Gifford[79]'s Weighted voting. These algorithms are for managing the distributed replicas of objects so that executions on replicas are equivalent to serial execution on single-copy object. This property is known as one-copy serialisable, and it guarantees the consistent state of the system.

Replicated objects are implemented by two kinds of sites: repositories and front-ends. Repositories provide long-term storage for the object's state,
while front-ends carry out operations. To execute an operation on an object, transactions must send an invocation to one of its front-ends. The front-end receiving the invocation tries to collect a sufficient quorum to carry out the operation. Therefore, a quorum for an operation can be defined as follows [Herlihy 86]:

**Definition of quorum:** Let \( Q \) be any set of replicated objects such that an operation \( q \) can be executed. \( Q \) is called the quorum of the operation \( q \).

A quorum may be divided into *initial* (read) and *final* (write) quorums. An operation reads the object's state from initial quorum, and results of the operation are written to the final quorum. A *quorum assignment* associates each operation with a set of valid initial and final quorums. These assignments are constrained by a *quorum intersection relation*: certain initial and final quorums are required to have nonempty intersections to ensure the single-copy property. An object's quorum assignment determines the availability of its operations; thus the constraints governing quorum assignment are the basic constraints governing the availability realised by a replication method. A variety of the Quorum-Consensus replication method is the Dynamic Quorum-Consensus replication method [Herlihy 87]. In this method, there is a dynamic quorum assignment algorithm to consider inaccessibility of replicas by failures and newly added replicas. Quorums for read and write operations are recalculated when a replica is added or deleted from the system.

### 2.7 Deadlock Detection

In modern computer systems, processes request services from each other. Upon requesting a service, a process enters a wait state if the service is not available. Waiting processes may not ever change their states because the requested resources are held by other waiting processes. This situation is called **deadlock** [Gray 79] [Elmagamid 86] [Coulouris 88].

Deadlock can be handled in three ways: prevention, avoidance, and detection. Prevention guarantees that deadlock can never occur in the system, avoidance detects potential deadlocks in advance and prevents
their occurrence; and detection allows the occurrence of deadlocks and then finds and resolves them.

Deadlock prevention can be achieved by the pre-declaration of all the required resources. All resources needed are reserved in advance. This scheme does not require run time support. The most important advantage of this scheme is that no transaction needs to rollback or restart due to deadlocks. However, it has two disadvantages. Firstly, pre-allocation of resources reduces concurrency of the system. Secondly, evaluation of the safety of the request causes additional overheads.

In avoidance strategy, transactions are allowed to proceed whenever the required resource is available. When a conflict occurs, transactions may wait to grant the request during a fixed time interval. If the resource is not available within the interval, the requesting or holding transaction may be aborted. Choosing the victim depends on the avoidance algorithm used. The major disadvantage is that transactions may be aborted unnecessarily. However, systems which support transactions already have the ability to abort transactions. Therefore, this scheme is more useful than prevention.

In a deadlock detection scheme, the requesting transaction can wait indefinitely to hold the resource. As a result, deadlocks may arise. Therefore, the system must detect and resolve the deadlocks. Usually, the detection algorithm finds cycles among transactions, each waiting for a resource held by one of the others. These waiting relations between transactions are generally considered as a directed graph. In the graph, vertices represent transactions and resources, and edges represent requests and allocations among transactions and resources. The main disadvantage of the scheme is the overhead due to detection of cycles in the graph and abortion and restart of transactions. Deadlock detection algorithms can be distributed or centralised. The distribution strategy may have additional overheads due to message transfers between computers. The correctness of a deadlock detection algorithm depends on two conditions. Firstly, all deadlocks must be detected in a finite time. Secondly, the detected deadlock must really exist. A deadlock which does not exist when reported by an algorithm is called a phantom or false deadlock. It may be detected if some computers in the distributed system hold out-of-date
information. Therefore, keeping consistent data is very important in deadlock detection algorithms.

2.7.1 Centralised and Distributed Deadlock Detection Algorithm

Processes in a distributed system execute on spatially separated computers and can communicate with one another by exchanging messages. There are two major differences between the distributed system and the traditional centralised system: firstly, the former lacks global shared memory to share a graph, and secondly, the delays in message delivery are not negligible. In principle, a protocol for deadlock detection in distributed systems is not different from a protocol for a traditional system. However, the distributed system constructs and maintains various graphs used for deadlock detection because it lacks global memory. To ensure that each site maintains accurate statuses of processes and resources, the concurrency controllers in each site exchange messages with each other. The messages cause additions and deletions of nodes and arcs in the graph. Arbitrary delays in message delivery cause out-of-order graph update. Because a single graph for deadlock detection is constructed and maintained in shared memory, out-of-order graph updates do not appear in the centralised system.

A centralised deadlock detection algorithm [Gray 79] requires that all information represented by the graph be kept in a single computer, which is responsible for running the deadlock detection and resolution algorithm. Whenever a process waits on an external resource, the central controller constructs a global and complete wait-for graph using information received from other controllers. Then, the controller executes the deadlock detection algorithm. The algorithm is simpler than a distributed one, and it does not detect false deadlocks. There are two main disadvantages of centralised deadlock detection: firstly, the high cost of constructing the wait-for-graph whenever a transaction waits on a global resource, and secondly, the vulnerability to failure of the central site.

In a distributed detection algorithm, no single site has all the information representing relations among all transactions and resources in the graph. Therefore, information in local sites must be passed between sites to detect deadlocks. Most distributed deadlock detection algorithms come in one of
two categories. Algorithms in the first category pass information for transaction requests to maintain the global wait-for graph. In the second category, simple messages are transmitted among transactions without construction of an explicit wait-for graph. However, a cycle in the graph is detected by returning a message to its initiator.

2.7.2 Distributed Deadlock Models

A distributed system consists of a set of processes which communicate with one another exclusively by messages and which may execute in different computers. The message communication protocol is assumed to be: any message sent by one process to another is received correctly after arbitrary but finite delay, and message transmissions obey the first-in-first-out rule. There are two deadlock models in message communication systems: resource and communication deadlocks. Resource deadlock model [Gligor 80] [Obermark 82] [Sihna 85-1] [Roesler 88] [Choudhey 89] is mainly used in transaction systems. In this model, deadlock occurs because processes may wait permanently for resources held by each other. Our distributed deadlock detection algorithm also belongs to resource deadlock model. The communication deadlock model [Chandy 83] is more general than the resource model. It is applicable to any message communication system.

2.7.2.1 Resource Deadlock Model

A resource deadlock problem arises in distributed databases. A distributed database includes resources, controllers, and processes. Each controller is associated with a set of resources which it manages and a set of constituent processes. A process can request resources located in the other controller as well as in its own controller. A process cannot execute unless it receives all the grant messages for the requests. A set of processes is said to be deadlocked when no process in the set can execute because each process requires a resource held by some other process in the set.

A process is said to be idle or blocked when it waits for a resource; it is said to be executing or active when it is not idle. Therefore, a process is permanently idle if it never holds a requested resource. If a single process runs in a distributed database, it will terminate in finite time and release
all resources. When more than one transaction run in parallel, deadlock may arise.

A process Pi is dependent on another process Pj if there exists a sequence of processes Pi, Pi1,...,Pim,Pj, where each process in the sequence is blocked and each process except Pi holds a resource for which the previous process is waiting. Process Pj is deadlocked if it is dependent on itself or on a process that is dependent on itself. In either case, deadlock exists only if there is a cycle of blocked processes each dependent on the next process in the cycle. As a result, resource deadlock detection algorithms declare that deadlock exists if and only if such cycles exist.

2.7.2.2 Communication Deadlock Model

An abstract description of a network of processes which communicate through messages is the communication model. In this model, there are no explicit controllers. The controllers are implemented by processes. Messages implement requests for resource allocation, cancellation, and release.

Every blocked process associates with a set of processes called its dependent set. A blocked process resumes upon receiving a message from any process in its dependent set; otherwise, it does not change its state or dependent set.

A non-empty set of processes is deadlocked if all processes in the set are permanently blocked. A process is permanently blocked if it never receives a message from any process in its dependent set. In other words, a non-empty set of processes is deadlocked if and only if all processes in the set are blocked, the dependent set of every process in the set is a subset of the set, and there are no messages in transit between processes in the set. A process is deadlocked if it belongs to some deadlocked set.

2.7.2.3 A Comparison of the Two Models

There are several differences between the resource model and the communication model. The most important difference is that in the communication model, a process must know from which processes it is expecting messages - it is waiting for these messages in order to resume
activity. Thus, the processes have the necessary information to detect deadlock if they act collectively. In the resource model the relation between two transactions is not directly known. It only knows which transaction waits for a given resource and which transaction holds the given resource. A controller at each site keeps information for its resource and only the controllers can recognise the relation between transactions. Therefore, the deadlock detector in the two models is different.

The other main difference is that in the communication model, a process can continue if it receives a message from any process for which it is waiting. However, in the resource model a process can proceed only if it holds all requested resources. In other words, the resource model is waiting for all resources and the communication model is waiting for any one message.

In the graph in the resource model, deadlock is detected by finding a cycle of processes, whereas in the communication model deadlock is detected by finding a knot of waiting processes. A knot is a vertex $k$ of a directed graph such that any vertices reachable from $k$ can themselves reach $k$.

2.8 Conclusion

In this chapter, we discussed topics related with our work. Current techniques do not utilise facilities of the new network environment, such as fast client machines, multicast, objects, etc, to improve their efficiency.

We will propose an object model, exploring the new environment, with a cache coherence protocol, a recovery scheme, and a deadlock detection algorithm. The object model utilises caches and multicast communication, and explores the role of client machines in distribution of work load.
Chapter III
Object Space

In this chapter, we describe an object space model. The model enhances the client-server model by using caches intensively and distributing the functionalities of conventional servers. The object space is divided into three sub-object spaces: private, active shared, and passive shared object spaces. Each shared object space is independent from other shared spaces. This division provides great flexibility and robustness without high overhead in comparison with the conventional client-server model.

3.1 Overview

We are proposing a persistent shared object space. It is used to store objects persistently and to share objects among users without explicit conversion of object formats by users. Shared object space partition (section 3.2), object naming (section 3.3), object grouping (section 3.4), and a standard object form (section 3.5) are used to improve robustness, efficiency, and user-friendliness.

The most different aspect of our model from that of other object stores, Mneme [Moss 90], ObServer [Fernandez 91], etc., is the partition of the shared object space. Each space can be implemented independently and the effect of a failure of a space on other spaces is minimised. We introduce active and passive object spaces. Active object space connects the passive object space with clients. There can be several active object spaces in a distributed system. Each active object space may be independent from a client's environment or it can be integrated with the client's environment. The former receives client's requests only by an independent message passing mechanism such as RPC. This kind of active object space can be used to provide a specific service rather than an
interactive service for a user. The latter is generally located with a client machine and communicates directly with that client's private objects. This kind of active object space is suitable for an interactive service and can improve the efficiency of the client system. Introduction of active object spaces makes our model flexible and easily expandable. An active object space can be used to provide a service or to connect a new language, such as Smalltalk, as long as it satisfies certain requirements (see section 3.2.2 and 3.3.3). Furthermore, its implementation is encapsulated; it can be altered without affecting other spaces.

Another aspect of active object space is that some of the workload of the passive object space management is distributed into active object spaces. The resulting functionality of the passive object manager is more or less the same as that of a conventional file system. The communication traffic between active and passive shared object spaces is kept to a minimum because client requests are processed in active object space, rather than forwarding them to passive object space.

Several different naming schemes are used in the entire shared object space. Active object space and passive object space use their own naming schemes. Active and passive shared object space consist of sets of (sub-)object spaces, and each (sub-)object space can use its own naming scheme. In other words, an object name in an (sub-)active object space may not be understood by another. However, all shared object space managers understand an independent naming scheme which is used for communication between object spaces: active object spaces, active and passive object space, or passive object spaces. Even though there is a name conversion penalty, this naming structure provides independence of one object space from another.

After termination of a process in a conventional computing environment, the data generated by the process are lost except for those saved in a permanent storage such as a file system. A persistent object space provides such a permanent storage. Unlike file systems, it eliminates explicit data conversion by the client when data in the space are retrieved. However such a conversion has to be provided by a system. Objects require conversions between volatile and non-volatile storage and for
communication between two nodes. Such conversions have to be simple and efficient. We propose a single form of object structure for both storage and communication.

3.2 The Object Space

The entities of a system should be identifiable within the system. When entities are identified by the same identification scheme and have the same meaning, the range of the identification scheme is called an address space. An address space will be called an object space if an identifier in the space represents an object.

![Diagram of conventional client-server model and object space model](image)

Figure 3.1 (a) Conventional client-server model and (b) our object space model

In the following paragraphs we contrast the use of a cache in the client-server architecture with its use in our object space model. Figure 3.1(a) illustrates the client-server model in which clients and server have their own processing spaces. To improve the efficiency, a cache is located in the clients' machine. A server has two major functionalities: providing a shared service and shared storage among clients. Caches are used to maintain replicas of parts of data stored in a server. The interface to
communicate with the server is provided in a client machine even though it may be transparent to the client.

The proposed object space model consists of several sub-object spaces as in figure 3.1(b). Each object space may or may not be shared among clients of a system. The former will be called Shared Object Space and the latter Private Object Space. Objects in an object space are active if they can process requests; otherwise, passive. A shared object space in which objects are active is called an Active Shared Object Space (ASOS), otherwise a Passive Shared Object Space (PSOS). A local identifier in an active (passive) shared object space is converted to a unique space-independent identifier. ASOSs and PSOSs comprise the shared object space in our model. The entire active (passive) object space consists of a set of independent active (passive) object spaces.

Each object space may have its own naming scheme and failures of an object space are localised. In other words, the object space can be heterogeneous. The active and passive object spaces together provide a conventional shared resource mechanism. PSOSs are responsible for basic persistent storage functionalities such as storing shared objects and recovering objects after system crashes. ASOSs support functionalities such as execution of objects, concurrency control, etc. This division of the entire shared object space makes it more flexible than the conventional client-server model in which caches and interfaces are commonly implemented in a system kernel. For instance, supporting active object spaces, which can be related with different languages, by one passive object space becomes easier by adjusting an ASOS for a specific language environment. For ASOSs, PSOSs are a kind of persistent object storage.

The client-server model has several disadvantages. First of all, the server machine must run fast enough to reponse clients's requirements quickly. The server process must provide both a running environment for several services and shared storage, therefore, a machine supporting such a process must be powerful to avoid a bottleneck at the machine. Second, a running environment provided by a server process is not very flexible. For instance, due to problems of consistency, the same concurrency control protocol must be used by services supported by the server process. Third,
caches are under-used relative to an object-oriented environment in which a cache can be improved to do more than just storing replicas of parts of data stored in a server. Fourth, communication traffic between client and server needs to be high to maintain cache coherence. When concurrent use may invalidate caches, updating all affected caches causes communication between caches and the server.

Our object space structure has several advantages relative to the client-server model. First of all, we do not require a very fast server. As mentioned above, the conventional server's functionalities are divided into ASOSs and PSOSs. Because ASOSs are mostly located in client machines and PSOSs do simple tasks for ASOSs, PSOS machines need not be as powerful as the server machine in the client-server model. Second, because objects are executed and several functionalities of a conventional server are supported in the ASOS, service request messages via the computer network can be reduced if the ASOS is located in the client machine. A cache coherence protocol for replicated services in caches is provided to maintain cache coherency with minimum overhead. Third, flexibility of the services is increased significantly. A service which requires a special environment can be built easily in an ASOS without affecting other shared object spaces. The service can use other services via the interface of the ASOS.

3.2.1 Passive Object Space

An object can be defined by three attributes: variables, operations, and context. Variables represent the state of the object and operations the mechanism to access and modify the state. The context contains its relationship with other objects, and its attributes such as running environment information, its string-oriented name, its owner, etc. The definition of an object can vary depending on the system. As a result, an object defined by one system cannot be used by another system. This system dependency can be eliminated by using a standard definition form.

Passive object space represents a truly system independent object space for several different languages (see section 3.5). Objects in PSOS are guaranteed their durability. As objects in PSOS are passive, they can be represented by streams of slots. A slot is the basic unit of flattened shared
objects and is the basic unit of communication between shared object spaces. Objects are created, removed, or updated by the manager of a PSOS. In addition, there are simple operations, used by the cache coherence protocol (see chapter 4 and 5), for version information to support concurrent sharing. Because the responsibility for concurrent sharing facilities and cache coherence is with ASOS, PSOS can be simple and efficient.

Even though PSOS provides a system independent definition of objects, the PSOS manager is not concerned with the structure of the objects it maintains. An object is stored in the same form as that in which it is sent by an ASOS. A system independent object is transformed into the standard form in the ASOS not in the PSOS. When an object does not require system independence, the objects are saved in a system dependent form, which reduces activation and de-activation time. This decision is made by the owner of the object, and a system dependent object can be transformed into a standard form later when this is required. ASOSs supply the inter-space object identifier (see section 3.3) and generally part of the state of the object to modify its state in the PSOS.

A PSOS consists of a volatile and a stable passive object space. The volatile space, a kind of working space, maintains objects before storing them in the stable space. The volatile space can be constructed upon main memory with disk but the stable space must be built on a stable storage (see section 2.5.1) which can recover its information after system crashes.

3.2.2 Active Shared Object Space

Objects are required to support certain functionalities. To respond to requests, objects must be active either in an ASOS or in private object space. An ASOS can accommodate several services or can be specialised for a service which requires its own running environment. A manager of an ASOS provides shared objects by importing them from a PSOS. When an object is created in an ASOS, the object must be stored by a PSOS in order to be shared.

Most ASOSs reside in client machines as caches. Caches are very important in terms of the efficiency of a distributed system. A cache is used
as a temporary storage containing parts of the information stored in a larger, slow storage. By keeping information in the client sites, communication delay, traffic, and access time are significantly reduced.

Due to rapid development of cheap, powerful workstations, most workstations nowadays have a 4-8 Mbytes of main memory and 40-80 Mbytes of disk storage. Therefore, maintaining a large cache space is not expensive nor impractical. Note that an ASOS is much smarter than ordinary caches; it only updates the state of objects in PSOSs when updating is really necessary and can determine when to do so. Another reason for placing ASOSs in client machines or any computer other than the PSOS is that the overall workload is distributed between the PSOS and ASOSs. This means client machines are more heavily used than in the conventional client-server model.

Apart from these functions, there are several further important aspects of our ASOS. First of all, by separating active and passive object spaces, the handling of objects in each space becomes much simpler. Secondly, any disruption of an ASOS does not affect other object spaces. Objects in PSOS or other ASOSs are not affected by the crash of an ASOS unless they are cooperating in their work.

3.2.3 The Required Architecture for the Object Space Model

Our model has three layers of object spaces. The basic layer, called the passive object layer, provides persistent object storage which is accessible by the second layer. The basic layer is a set of PSOSs. The second layer, called the active object layer provides working spaces for objects stored in PSOSs in the passive object layer and is a set of ASOSs. The top layer is called the private object layer and is a set of private object spaces.

A service is a set of groups of objects which cooperate to provide a facility or facilities. The passive form of these objects is stored in a PSOS or a set of replicated PSOSs. To provide the facility (facilities) of the service, a replica of the service must be loaded into at least one ASOS. However, the service need not be loaded entirely.
In our model, all object migration is done in passive form. We will use passive migration. By migrating passive objects, we eliminate many problems related with active object migration (see chapter 2.2.2).

Figure 3.2 shows the architecture of our model. A user of node 1 can use object A and four services. Service 1, service 2, and service 4 are active but service 3 is passive. The ASOS in node 3 is shared by other ASOSs in node 1 and node 2. The user in node 1 has two choices for using service 4. One is using service 4 in node 2 and the other is loading service 4 from node 5. The former is used by remote invocation and the latter by replication. If the service is too big to fit in the ASOS in node 1 or requires a special environment, such as a highly protected one, then the user must access the service via remote invocation.

Figure 3.2 The Architecture of our model
Objects in private object space can communicate with objects in an ASOS but not with objects in a PSOS. Only objects in an ASOS can communicate with both private and passive objects. We provide an ASOS for each private object space. As mentioned before, the ASOS acts like a cache and connects the private object space with the rest of our model. An ASOS can be specialised to integrate with a private object space. For example, in Smalltalk [Goldberg 83], the ASOS can be a set of objects whose operations can be executed by the interpreter which executes operations of private objects as well. This arrangement reduces communication overhead between private and active objects; message passing is done within a process rather than by inter-process communication.

All PSOSs have a standard interface definition and active objects in ASOSs use the operations in the interface to access and update corresponding passive objects in PSOSs. The reason for requiring the standard interface to operations is that passive objects in PSOSs cannot respond to any message sent to them. Because all objects have the same operations, delegated objects rather than the passive objects themselves can handle all incoming messages on behalf of passive objects. These delegated objects comprise what we call a passive shared object space manager.

The structure of the passive object layer depends on the required degree of availability of the system. If a system does not require high availability, stable storage (see section 2.5.1) with recovery information can be used to construct PSOSs. In this case, the result of a single processor failure is that services stored in the node are not accessible until the crashed node is recovered from its failures. However it provides sufficient durability.

When a system requires high availability, services can be replicated among PSOSs to prevent a total loss of services due to a node failure. For example, the Quorum-Consensus method [Herlihy 86] can be used to access services in this kind of structure. Because services in PSOSs are replicated into ASOSs, availability in our model is further increased relative to non-replicated systems. Note that cache replication is independent of the structure of PSOSs.

There are two major differences between replication into ASOSs from PSOSs and among PSOSs. Firstly, ASOS caches are not required to replicate
As in Figure 3.3, several different AIDs can exist in several ASOSs and PIDs in PSOS. This reflects the flexibility of our object spaces; the naming structure of an object space can be independent. We will discuss string-oriented naming for AID in section 3.3.3 and fixed-size identifiers for AID in section 3.3.4 and for a PID in section 3.3.5. These can be seen as examples of each of the identifiers. Note that there is no restriction on the use of any form of identifier as the naming scheme of an ASOS (PSOS) as long as the ASOS (PSOS) manager can convert their AIDs (PIDs) to IOPs and vice-versa. We assume that objects have their string-oriented name as one of their attributes but this is not compulsory.

Several heterogeneous PSOS managers may comprise the entire passive object space as long as they understand IOPs. This gives freedom of implementation of a PSOS - each PSOS has to satisfy pre-defined specifications but its implementation is up to its creator. Both AIDs and PIDs are mapped into corresponding IOPs. In the model, an IOP of an object cannot change but its AID and its PID may change. The internal change of identifier does not affect other object spaces as long as new AIDs or PIDs are mapped to the IOP; other object space managers do not notice this change. As a result, re-organising an object space becomes easier.

In an object-oriented environment, objects are related to each other. A service represents a set of groups of objects which co-operate to provide a facility (or facilities). Grouping objects depends on their relationship with other objects in the service. Grouping will be explained later. The name of an object consists of a service name, a group name, and an object name.
To increase the speed of the name conversion, we prefer a structured naming scheme to a plain one.

3.3.2 The Inter-space Object Identifier

Between shared object spaces, objects are referenced by fixed size numerical object identifiers, called Inter-space Object Identifiers (IOP). IOPs are required to be global and unique over the entire shared object space. Any object managers of ASOS or PSOS have to provide a mechanism for converting their objects' identifiers to IOPs and vice-versa.

<table>
<thead>
<tr>
<th>A Service Identifier</th>
<th>A Group Identifier</th>
<th>An Object Identifier</th>
</tr>
</thead>
</table>

Figure 3.4 The structure of an inter-space object identifier

An IOP consists of three fields: a service identifier, a group identifier, and an object identifier - see figure 3.4. The service identifier represents a service (e.g., a text editor). The group identifier is used to group objects. Object identifiers indicate objects in the group. The size of an IOP is 64 bits: 24 bits for the service identifier, 16 bits for the group identifier and 24 bits for object identifier. The group and object identifiers are only unique in a service but not across services. Objects can be referenced with only object identifiers within a group and with group and object identifiers in a service in PSOSs. This reduces the size of the flattened object.

Each service has one reserved IOP. This IOP represents the entire service. When a client wants to use a service, it needs to know the reserved IOP of the service. The reserved IOP has its group and object identifiers set to zero. All reserved IOPs are stored in a binding service. One of the well known methods may be used to locate the binding service.

IOPs are introduced to enhance the flexibility and expandability of our model. Neither ASOS nor PSOS are restricted in their internal structure except for the following few conditions: understanding IOPs and including certain basic objects (see section 3.4). For example, a new system, such as
Smalltalk [Goldberg 83], may want to use an existing service. The only thing to do is to build a customised ASOS. The ASOS understands Smalltalk messages and connects with the rest of the model.

3.3.3 A User Friendly Active Object Identifier

Two kinds of naming schemes can be considered as objects' identifiers in object spaces. One is user-friendly and the other is numeric naming.

A user friendly naming scheme is appropriate in private object spaces in which the use of symbolic names is preferred to the use of numerical identifiers. Even though ASOSs are logically different object spaces from private object spaces, some ASOSs are related with private object spaces.

When ASOSs can be accommodated in a client machines, they have their environment customised to that of the private object spaces for users' convenience. They are used to connect private object spaces with shared object spaces. Using the same naming scheme for these ASOSs as that of the private object space is desirable to enhance user-friendliness and to provide name transparency of the system.

The managers of such ASOSs must perform all the usual functions, including the provision of a conversion scheme between user-friendly identifiers and IOPs. Most systems/languages such as Smalltalk [Goldberg 83], Arjuna [Dixon 89] [Parrington 88, 89], etc. use strings to identify their objects at the user level. Therefore a string-oriented naming scheme would normally be appropriate.

The name of an object in an ASOS is called an active object identifier (AID) - see figure 3.5.

<Service Name>.<Group Name>.<Object Name>

e.g.) System.Collection.anArray

Figure 3.5 Three-level naming scheme in ASOS
When a shared object is created or imported from other object spaces, it may cause a name conflict with an existing object. This problem must be solved in a shared environment. Service names are unique in the distributed system and group names and object names are unique within a service; service names together with the locations of the corresponding services are registered in the binding service to make them available for clients. Due to the service name component of an object, objects belonging other services cannot conflict with the object.

3.3.4 Specialised Active Object Spaces and Their Identifiers

An ASOS can be built to provide a special service for the other ASOSs. In this circumstance, the naming structure can be specialised to improve the efficiency of the service. For instance, an ASOS could provide a binding service. Its task it to find the locations of servers of required services for clients. A binder can use the IOPs of services to indicate the locations of PSOSs which store these services. In this case IOPs become keys to the location objects in the binder.

Another use for this kind of ASOSs is for connecting with services in other systems - figure 3.6. In this case, a system connected via such an ASOS is considered as the private object space of the ASOS. The ASOS acts as an interface between our model and the system it connects with. The ASOS must understand the naming of both systems. This is simplified by using global identifiers of the other system as its AIDs. In our model the ASOS makes services in other systems accessible to users. Any change made in the other system does not affect the components of our model apart from the connecting ASOS - because the implementation of the ASOS is hidden from the rest of system, it does not affect any other object spaces.
3.3.5 A Passive Object Identifier in a PSOS

A PSOS has its own naming structure. Identifiers of objects in the PSOS are called Passive object identifiers (PID). The structure of a PID can vary between different PSOSs. But with a PID in a PSOS, the manager of the PSOS can find the object and its attributes indicated by the PID. In figure 3.7, an example of an implementation of a PSOS is shown. A PID in the PSOS is a tuple, \(<\text{Service table index, Segment table index, object table index}>\). As a PID is set of indexes, relocation of the object does not affect the PID.
3.4 Object Grouping

An object uses other objects to complete its operations; whenever the object is used, some of these objects are required. A service generally has many functionalities and each functionality is usually supported by several objects. Therefore, grouping objects supporting a functionality is reasonable. In addition, some objects in the shared object space can be used commonly by all shared services. These kinds of objects can be grouped together depending on their relationship. By grouping objects, the waiting time of client processes, workloads of passive object managers, and communication traffic can be reduced.

3.4.1 Object Structure

An object can be defined as a set of operations and a set of variables, which represent the state of the object. Note that in an object-oriented environment, variables representing the object state are also objects.

There are two situations in which a shared object must be flattened: when an object moves via a network and when an object is stored on the secondary storage. The flattened object may be in a system/language
independent form or dependent form. Unless the owner wants to support different systems, objects in PSOS have a system/language dependent form to increase handling efficiency. Because objects are passive in PSOS, the PSOS manager does not need to unpack the flattened objects, and it can supply the flattened object to an ASOS directly. This reduces the workload of PSOS managers. If the format is the standard one, the ASOS manager must convert it to the system dependent form when it activates the object and reconvert it into the independent form when it is deactivated.

3.4.2 Object Relationships

In an object-oriented system, an object is related with other objects in an object space. When an object needs to move into another space, related objects have also to move with it because the moving object can be useless without these objects. For example, the state of an object is a set of instances of other class objects. If those class objects do not exist in the object space, then the object can either import its classes or use remote active classes. But the second choice can be too expensive in terms of message passing cost if they are located in other machines. Remote referencing is more expensive than local referencing. It requires message passing and flattening the arguments is a very expensive operation. Loading instances with their classes is a better choice because avoids remote referencing. Another important thing to consider is the environment of an object. An object, defined in a particular language, cannot be executed by another language.

Figure 3.8 shows an example of object relations. The object I is an instance of a class object C2 and the class C2 is a subclass of the basic class B1. The basic class, a kind of basic objects, will be explained in section 3.4.3. The instance object I has two variables, S1 and S3. These variables themselves are instances of class C1 and C3, respectively. Therefore, the instance object I has a relationship with C2 and B1 via class hierarchy and with C1 and C3 via its variables.
An object is, firstly, related with hierarchical objects, its class and superclasses; it can use all operations in its class and superclasses. Secondly, classes of its instance variables have to accompany it together to avoid remote invocation. Without these classes, instance variables cannot execute any message.

Variables can be categorised into several types. For instance, in Smalltalk, there are four kinds of variables which can be used by an object: global variable, class variable, instance variable, and temporary variable. Global variables are available to all objects in an object space. Global variables can be grouped together and the group may be duplicated into different object spaces. Class variables are shared by instances of a class. Instance variables are private objects of an object. Temporary variables are used to save temporary information by a method. All those variables, their classes and superclasses can be related with an object.

To summarise, an object has the following relations:

1) Class hierarchy
2) embedded objects: variables
3) attributes of an object space in which the object exists
Embedded objects themselves are objects: they have the above relations themselves. As a result, the total relation diagram of an object becomes a graph. We use cluster to trace all objects related with an embedded object. The definition of cluster "*' is that the cluster, $V^*$, of an object set $V$ is an object set, $V^n$, where $V^n$ is defined by:

\[
V^0 = O \\
V^1 = V^0 \cup R(V^0) \\
\vdots \\
V^n = V^{n-1} \cup R(V^{n-1}) \\
\text{until } V^n = V^{n-1}
\]

where $O$ is a set of initial objects, $R(S)$ is a set of objects related with objects in the object set, $S$, and $R(\text{primitive objects}) = \emptyset$.

The relation, $R$, for an embedded object is its instance variables and their classes.

3.4.3 Logical Object Grouping

As mentioned above, there are several relationships among objects. To make the object useful, moving an object means moving the object and related objects with it. A service usually consists of several objects and their related objects. Those objects together support all functionalities of the service. However, the entire functionality of the service is not always required; objects and their related objects, supporting the required functionalities, must be loaded into the ASOS.

Objects supporting a service are grouped according to how close their relation with other shared objects. For instance, some objects are accessible by any user but others by only the owner of the service. Those objects should be separated. Moreover, an object needs several other objects to complete its functionality. Those objects should be grouped together. Unfortunately, the grouping of objects is application dependent. Therefore, the owner of a service must provide grouping information when the owner creates the service in the shared object space. However, grouping can be helped by the system. The following things are not required to be mentioned by the owner of the service:
1) superclasses of an object,  
2) classes of its instance and their superclasses,  
3) basic system objects,  
4) attributes of the object space

but the following things must be mentioned

1) the service name,  
2) group names of objects,  
3) access right of objects which do not have the default access right,  
4) group relationship if necessary, and  
5) whether basic private objects are the same as the basic system objects.

There are two levels of grouping: service and object group. A service represents a big functionality and object groups sub-, self-contained functionalities of the service. A group relationship is used to load related groups in advance. If two groups are tightly related, the other group can be loaded together when the first group is loaded.

In an object space, many objects are commonly referenced by other objects. As a result, these objects may be included in almost all services. There are several advantages to grouping these objects into an independent group. First of all, we can assume that these objects already exist in ASOSs. It improves the process of finding related objects greatly. These objects serve as the terminal position of any tracing procedure. Secondly, by excluding these objects from services, a lot of storage space in PSOS is saved. Thirdly, the length of communication packets between object spaces is reduced. We call this group of objects the Basic Object Group. Objects in basic object group are called basic objects and class objects are called basic classes. A basic object which is atomic (see section 3.5.2) is called a basic atomic object. Objects belonging to the basic object group may vary depending on the system in which the object space exists. A set of <service, object group> pairs can be used to represent simple objects such as True, False, Characters, etc.

To find out all the objects related with a service, we introduce a tracing algorithm. A service without grouping can be traced by the following algorithm:
1) \( \text{Srv}_0 = I(S_0) \cup C(S_0) \cup C(I(S_0)) \cup S(C(S_0) \cup C(I(S_0)))^* - B \)
2) \( \text{Srv}_i = \text{Srv}_{i-1} \cup V(\text{Srv}_{i-1}) \cup C(V(\text{Srv}_{i-1})) \cup S(C(V(\text{Srv}_{i-1})))^* - B \)
3) Repeat step 2) until \( \text{Srv}_i = \text{Srv}_{i-1} \)

where \( \text{Srv}_i \) is a set of objects which support the service, \( \text{Srv} \). \( S_0 \) is the set of objects supplied by the creator of the service, \( \text{Srv} \). \( \text{Srv}_0 \) is the set of objects which is directly related by objects in \( S_0 \). By step 2) \( \text{Srv}_i \) contains embedded objects by objects in \( \text{Srv}_{i-1} \). \( I(S) \) is the set of instance objects in a set, \( S \), of objects, \( C(S) \) the set of class objects, and \( S(S) \) the set of super classes of a class object set. Let \( S \) be a set of objects. \( V(S) \) is the set of variables representing state of objects. For instance, in Smalltalk, \( V(S) \) is \( V_i(S) \), \( V_T(S) \), \( V_c(S) \), \( V_g(S) \). \( V_i(S) \) is a set of instance variables, \( V_T(S) \) is a set of temporary variables used to save temporary state of object to process messages. \( V_c(S) \) for class variables and \( V_g(S) \) for global variables. \( S(S)^* \) is the set of cluster of superclasses of objects. \( B \) is the set of basic objects defined in the object space.

Tracing groups of objects is similar to tracing objects of a service. The group tracing algorithm is:

1) \( \text{Grp}_0 = I(G_0) \cup C(G_0) \cup C(I(G_0)) \cup S(C(G_0) \cup C(I(G_0)))^* - B \)
2) \( \text{Grp}_i = \text{Grp}_{i-1} \cup V(\text{Grp}_{i-1}) \cup C(V(\text{Grp}_{i-1})) \cup S(C(V(\text{Grp}_{i-1})))^* \)
   \( -(B \cup \text{Grp}^\dagger) \)
3) repeat step 2) until \( \text{Grp}_i = \text{Grp}_{i-1} \)

where \( \text{Grp}^\dagger \) is a set of objects belonging to other group of the service. The only difference is removing objects which already belong to \( \text{Grp}^\dagger \).

The service with grouping is simply a set of groups traced by the above algorithm:

\( \text{Srv} = \cup \text{Grp}_i \) where \( 0 \leq i \leq n \), \( n \) is the number of groups in the service.

3.4.4 Physical Object Grouping

Object groups can be divided into smaller groups by the object space manager to improve the efficiency of the system. Firstly, objects having
different attributes, such as different access rights, can belong to the same logical group. Secondly, overlapped objects which belongs to several logical groups, can form another group themselves.

A group of objects in PSOS is called an object segment. Even though an object segment is related with a (logical) object group, one logical object group can be stored in several object segments or vice-versa. When an object is requested by an ASOS manager, the entire object segment is sent rather than an object. Sometimes, several object segments can be sent if they are closely related. By sending segments rather than single objects, the number of object faults (see section 3.5.4) can be reduced - reducing response time and network traffic. Furthermore, an object segment can be used to control access rights of clients. Even though each object may have its own access rights, an object group is usually required. Therefore, an object segment is a more suitable unit to control accessibility. How to load objects having different access rights is described in section 3.5.4.2.

3.5 Object Structure and its Transfer

In this section we describe problems and solutions involved in object movement. Moving an object requires careful consideration such as its state in a computer, its structure, and relationship with other objects.

3.5.1 Process and Object Transition States

To execute an object in an object space, two elements must be considered. One is a process and the other is a context or more generally an environment. An ordinary object does not have the ability to execute its own operations - it is not self-executable. Only processes have such an ability. We distinguish a process as self-executable.

\[
\text{Object} = \text{state} + \text{behaviour}
\]

\[
\text{Process} = \text{self-executability}
\]

A process is generated by request of the system or users. A process is an object; it has its own state and behaviour. But these are related with the underlying system rather than ordinary objects which users see. More importantly, the process is attached to an ordinary object by its creator, in
which it finds its task. It does not interfere with object's state to which it is attached. The object's state means the state represented by its variables. The process changes the object's state only by instructions described in the object. Therefore, for objects, a process is no more than an agent which gives executability to it.

![Diagram](image)

**Figure 3.9** The states of a) a process and b) an object

A process has three transition states: ready, running, and waiting and an object has four states: flattened, ready, running, and waiting. In Figure 3.9, the states of process and objects are shown. The flattened state of an object is a linear object structure which can pass through computer network links and be stored on the computer storage without further processing. A process in the ready state is ready to execute an operation of an object. An object in the ready state is ready to accept processes, i.e. to have its operations executed. A process in the running state is executing an operation and an object in the running state has at least one process executing its operations. A process or an object in the waiting state waits for a required event. A process can change its state by itself but the object
requires a process to change its state. For instance, an object in the ready state requires a process to be in the running state. Objects in the passive state require another object, whose operations can interpret information in the flattened object. The state of an object is seen by a process rather than the object that actually has such a state. For example, an object can be in a running state for one process but in a waiting state for another if two processes can execute operations of an object concurrently. In other words, for each process, the current state of the object may differ at the same time because each process can change the state of the object independently.

There are two types of objects: serial and concurrent. An object is a concurrent object if it can accommodate more than one process at the same time; otherwise it is a serial object. Concurrent objects have concurrency control to maintain their consistent states and they have multiple states; one for each process.

A process uses several objects during its existence. When a process executes an operation of an object, the operation may require the use of operations of other objects. Such a request is done by message passing in an object-oriented environment. The sender is suspended - the sender changes its states to wait and the process moves to the receiver.

A computer has at least one process, called the operating system, and usually more than one process exists in the computer. Therefore, a process related with an object does not need to move together with the object because a process can be obtained in the destination computer. For example, a transaction object with a process can change states of objects consistently. Without the process, the transaction object cannot start its operations. After finishing all operations of the transaction objects successfully or being aborted altogether, the process terminates.

Objects need an environment in which to run. For instance, a transaction object requires at least one process to run and other objects which can unflatten and flatten objects and which control the communication between computers if the transaction object uses shared objects. Such an environment related with an object is called the context of the object. A context contains pre-requisite objects; system and computer types, whether the object is serial or concurrent, and whether the object requires a process.
3.5.2 The Structure of Atomic Objects

Shared objects are divided into sequential objects and concurrent objects. Sequential objects can be used by one transaction at a time and concurrent objects can be used by several transactions concurrently. Sequential objects can be seen as coarse-grain concurrent objects because the whole objects are locked by a transaction.

![Diagram of atomic objects](image)

Figure 3.10 (a) an atomic object structure for the active object model
(b) an atomic object structure for the passive object model

There are two access control methods for objects: a message queue or a monitor. The former is useful for the active object model [Agha 86] and the latter for the passive object model. In other words, an object in the active object model has self-executability - a process is attached to each object; it can handle a message by itself, but an object in the passive object model cannot; it requires a process to do it for the object. When an active object tries to use an operation in another object, it sends a message. The message is queued in the receiver as in Figure 3.10 (a). It is processed by the receiver and its result is returned to the sender. Actor [Kafura 89] is the best-known example of this kind. If an active object suspends its operation because the operation may cause inconsistent state, incoming messages in the object's queue are not processed. As a result, all objects that had a message in the queue are also suspended. Meanwhile, a passive object has
no way to process incoming messages. Therefore, the sender is suspended, and then the process tries to execute the operation of the receiver. However, the receiver object requires access control to prevent inconsistent state. The receiver object forces the process to execute a monitor before they access the receiver object. Whether their accesses are granted depends on the state of the monitor. The monitor blocks any access which may cause an inconsistent state, or it permits all access but undoes effect of accesses which cause inconsistent state.

In our model, we use the passive object model. A monitor is related with an object. When a process activates an object, the monitor is activated before the process accesses the actual object. The monitor has two states: shared and exclusive. When the monitor is in the shared state, a process can access the object; otherwise the access is not permitted and the process is blocked. So far the monitor acts like a lock. However, the monitor permits certain system processes to access its state even though its state is exclusive. In other words the exclusive mode means exclusive to other ordinary process such as a process belonging to a transaction. Such an access may result in the abortion of the transaction which transfers the shared state of the monitor to the exclusive one.

Processes related with a recovery object, a deadlock detector, and update objects are examples of system processes. The monitor itself is an object, therefore it has its own state. The state may include the intention lists of transactions, version information, and a wait-for graph of the object. These states are related with the policies which are employed by the system, i.e., which kind of concurrency control, deadlock detection/avoidance and recovery techniques used in the system. The monitor provides all operations to restore the state of the object and to abort a transaction. Such operations are provided by an inheritance mechanism; all concurrent objects are subclasses of a monitor class. To prevent concurrent usage of sequential objects in concurrent environments, a sequential monitor can be used. When shared objects are loaded by a request of a transaction, all monitors of these objects provide exclusive access to the transaction.
An object is atomic if the object has a monitor and its state objects are also atomic. If an object has a monitor and it does not have any other object as its state, the object is called a basic atomic object. An integer is such an object. A shared object in a PSOS does not have a monitor because it is not used concurrently, but replicas of the object are required to have their monitors in ASOSs. Many functionalities such as concurrency control, transaction atomicity and deadlock detection are managed by the ASOSs. Moreover, all information to support such functionalities are in monitors. Like processes monitors are also created in ASOSs when shared objects are loaded.

3.5.3 A Standard Form for Flattened Objects

A PSOS can serve several different ASOSs. As mentioned earlier, each ASOS may support a different system. Then, an object generated by an ASOS may be meaningless for another ASOS. As a result, we propose a standard form for flattened objects which can be used by several different ASOSs. Actually, the manager of a PSOS is not concerned with the structure of flattened objects. Managers of ASOSs interpret flattened objects; managers recognise and generate flattened objects in this standard form. The definition of our standard flattened object is:

\[
\text{<Object> ::= <SystemTag> \{ <ImmediateObject> | <SimpleObject> | <ComplexObject> \} \}
\]

\[
\text{<ImmediateObject> ::= <ClassTag> <ImmediateValue>}
\]

\[
\text{<SimpleObject> ::= <Size> <SimpleBody>}
\]

\[
\text{<SimpleBody> ::= <ClassID> \{ <Slot> \}*}
\]

\[
\text{<ComplexObject> ::= <Size> <ComplexBody>}
\]

\[
\text{<ComplexBody> ::= <ClassID> \{ <ImmediateObject> | \{ <ReferencePointer> \} \} \}
\]

\[
\text{<ReferencePointer> ::= <SystemTags> <InternalPointer>}
\]
Simple and complex objects have size and class identifiers. The size field [28 bits] has the size of objects counted in Slots. A Slot [4 bytes or 32 bits] is the basic unit of the storage. Therefore, the maximum size of an flattened object is 1 Gbytes \[2^{30} = 256 \text{ Mbytes} \times 4 \text{ bytes}\]. The value in the size field calculates the size field itself and the Class ID field as well as the size of the object body. Then, the class identifier of the object follows. The actual class identifier is an IOP of a class object, so its size is 64 bits. Most class objects of instances are in the same service. The class identifiers may be reduced but some class objects do not belong to the same service. They require IOPs. The main difference between simple and complex objects is that the components of a simple object are all the same kind of objects. A complex object may contain more than one kind of objects. Those objects are indicated by reference pointers.

Reference pointers are introduced to avoid nested packing of objects (children objects) within the packed object. Otherwise when the size of a nested object (child object) is changed, the affected object (parent object) must be repacked rather than simply exchanging its reference pointer of the nested object with that of the new one. In addition, reference pointers with system tag filled make garbage collection easier. The structure of a complex object is simple; its body contains either immediate objects or reference pointers, and both are the fixed size. Therefore, a garbage collector can mark objects using system tag filed whether they are visited. The system tag indicates whether the collector requires further trace. If the tag is either immediate or simple objects then more tracing from these object is not required. A complex object is the only object that requires further tracing. Then, the collector knows how many objects are referenced by the object via the size field of the object, and then finds out where these object are via the reference pointer.

Figure 3.11 gives an example of an object and its standard form. An object A has two instance variables: a pointer and an object B. A pointer has two integer values as its state. The object B does not belongs to this service; for instance, it may be an basic object.
<System Tag> := [4 bits] : for pre-defined system tags
  0000 : immediate object
  0001 : simple object
  0010 : complex object
  0011 : reference pointer
  0101 : big object whose size is greater than 16M slots
  1xxx : referenced object (for garbage collection)

<Class Tag> := [4 bits] : for pre-defined basic objects
  0000 : small integer [16 bits]
  0001 : character
  0010 : boolean
  0011 : null pointer

<Size> := [28 bits] : the size of the object
<Class ID> ::= [64 bits] : the IOP of a class object
<Immediate Value> ::= [24 bits] : the value of an object
<Internal Pointer> ::= [28 bits] : the location of an object within a flattened object

There are three kinds of objects: immediate, simple and complex objects. Immediate objects represent very simple objects such as booleans, characters, and small integers which are basic objects in most systems. All immediate objects are simple objects. These objects are used to save storage space. A simple object is an object which has homogeneous structure such as an array of an immediate objects, a string, a long integer, a real number, etc. Complex objects are constructed from a set of objects which can be simple or complex objects. All objects except immediate or simple objects are complex.

All kinds of object have a 4 bits system tag. The system tag is used to indicate object categories and to maintain objects. The first eight bits, 0-7, are reserved and others, 8-15, are not used. An immediate object has a class tag and a value field. The class tag is used to indicate class or type of immediate objects. Both system tag and class tag have default values.
a) An object structure in an ASOS

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[4 bits]</td>
<td>[28 bits]</td>
<td>[64 bits]</td>
<td>[4 bits]</td>
<td>[28 bits]</td>
<td>[28 bits]</td>
</tr>
<tr>
<td>0010</td>
<td>5</td>
<td>xxxx ... xxxx</td>
<td>0011</td>
<td>bbbb...bbbb</td>
<td>0011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sys. Tag</th>
<th>Size</th>
<th>Class ID</th>
<th>[Sys Tag</th>
<th>Int. Ptr 1</th>
<th>[Sys Tag</th>
<th>Int.Ptr 2</th>
<th>Ref. ptr for object B</th>
<th>Ref. ptr for a pointer</th>
</tr>
</thead>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>pppp ... pppp</td>
<td>0010</td>
<td>5</td>
<td>yyy ... yyy</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bbbb ... bbbb</td>
<td>0001</td>
<td>5</td>
<td>zzzz ... zzzz</td>
<td>xxxxxxxxxx xxxxxxxxxx xxxxxxxxxx xxxxxxxxxx</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sys. Tag</th>
<th>Size</th>
<th>Class ID</th>
<th>The Inter-space identifier of object B</th>
</tr>
</thead>
</table>

pppp...pppp : the location of the pointer at the flattened object A
bbbb...bbbb : the location of object B at the flattened object A
Ref. ptr (reference pointer) = [ System tag | Int. ptr (Internal pointer) ]
Sys T. : System Tag, Cls T. : Class Tag

b) The flattened object of the object in a)

Figure 3.11 An example of an object and its flattened form

3.5.4 Crossing space boundaries

Object movement between different object spaces requires careful consideration. An object is related with other objects though inheritance and instantiation. Inheritance allows objects to share behaviour and
instantiation is creating an instance of a class object. Moving an object without its related objects causes considerable system overhead due to frequent object faults (see section 3.5.4.2) or remote invocations. We will explain two important boundary crossing cases: private object space to ASOS and between ASOS and PSOS.

3.5.4.1 Moving Private Objects into an ASOS.

Migrating an object from a non-shared object space to a shared object space is equivalent to creating a new shared object, which is an exact copy of an object in non-shared object space, in a shared object space. This process involves identifying objects which have to move into shared object space and generating new object names. The former is due to the relationship among objects and the latter is to avoid name conflict and to identify them in the shared object space.

The object becoming a shared one may depend on several private and shared objects. Because these objects are related with functionalities of the new shared object, related private objects have to be shared as well. These objects are copied and assigned new names. All objects referenced by the shared object directly or indirectly are related objects but it may include too many objects, thus the number of related objects must be decreased. First of all, there are common objects, called basic objects, which are frequently used by objects. These objects can be excluded from the related objects. Whether a basic system object is the same in the private object space and the ASOS must be indicated by the owner of objects. Secondly, shared objects which belong to another service are excluded. Those objects are indicated by their IOPs.

Private objects are copied into an ASOS and their behaviours and states are fixed. Changing of the behaviours and states of original objects in a private object space does not affect corresponding objects in the ASOS. In other words, they are considered different objects. As a result, newly moved objects must have new identifiers, and these identifiers should not conflict with the existing ones. There is an identifier generating service, called IOP generator. Each service has its own IOP generator, an instance of IOP generator. The IOP generator creates a service identifier, and then group identifier and object identifier. A service identifier must validate its
loaded when the service is required. Missing objects are replaced by special objects called *demanders*. When an object requires a missing object by sending a message to the demander, the demander takes one of two procedures: loading the requested object or refusing the message sent the object. The latter can happen when the user has no access right over the object. The requirement of a missing object is called an *object fault*. Objects are transferred from PSOS in passive form; the object can do nothing until it becomes active. Therefore, there must be an activator in an ASOS for those objects. The activator checks all IOPs in the objects, and then generates corresponding AIDs. If there are unresolved IOPs then it creates object demanders for them. The activator saves the <AID,IOP> pairs into a system table to use them when the object is deactivated. When the service is unloaded, these pairs are deleted from the mapping table.

3.5.5 A Communication Protocol

Communication between shared object spaces is done by message passing. The communication media are reasonably reliable; media failure is very rare and messages can be occasionally lost but the probability of a loss is low. We do not consider media failures in this thesis - media failures block communication and are silent. We assume that any failed node is silent.

In a local area network, group communication is efficiently achieved by *multicast* such as BSD 4.3 UDP/IP multicast protocol implemented by Steve Deering [Deering 89] at Stanford University. Multicast delivers messages to multiple receivers which belong to a certain group. A set of shared object spaces which provides the same service is called a *cluster*. Any active service has at least two members in the cluster: the ASOS in which it is running and the PSOS in which it is saved.

Multicast used in this model is not required to be reliable except for guaranteeing that all messages sent by the ASOS must be delivered to the PSOS. In other words, the ASOS requires a positive acknowledgement from the PSOS but does not require any acknowledgement from other members of the cluster. If the manager of the PSOS receives the same message more than once, it discards duplicated messages. As soon as the ASOS (actually objects in the ASOS) receives the acknowledgement from
the PSOS, it continues its processing. By losing messages to other members of a cluster, the object related to the missing message remains in an inconsistent state. However, the probability of losing update messages will be very small and the situation will be corrected without re-loading the entire service from the PSOS. Moreover, the inconsistent state affects only objects located in other members of the same cluster and any update requests from these members will be rejected by the PSOS. Note that a cluster is related with one service rather than a shared object space and a shared object space can have several clusters in it. Objects belong to other clusters in the same computer are not affected.

Each shared object space has a cluster list. Each element of the cluster list has locations of ASOSs and the PSOS. When an ASOS loads a service from a PSOS or a member client disposes the service, then the member of the cluster of the service are changed. Losing membership information may affect the consistency of the service. How to cope with the inconsistent service is described in chapters 4 and 5.

3.6 Summary

In this chapter we proposed an object space model which is used throughout the rest of this thesis. The object space model enhances the client-server model for use in an object-oriented environment. The whole object space is divided into three object spaces: private object space, active shared object space (ASOS) and passive shared object spaces (PSOS). A private object space maintains private and non-sharable objects in a user's machine, and ASOSs and PSOSs keep sharable objects. ASOSs provide a running environment for active objects and cache coherence in the context of transactions. PSOSs provides permanent storage for passive objects.

A user friendly naming scheme is used in some ASOSs which are integrated with private object spaces to provide naming transparency, and other ASOSs may use numerical naming to improve reference efficiency. However, naming schemes used in ASOSs (PSOSs) may vary according to the requirement of the services running (or stored) in the ASOSs (PSOSs). To communicate with each other, ASOSs and PSOSs are required to understand a common identifier, the inter-object space identifier (IOP).
uniqueness to avoid name collision with other service ones. Group and
object identifier are required to be unique only in the service, so the
validation can be done by the IOP generator. Note that until those objects
are moved into PSOS, they are neither truly shared nor persistent.

3.5.4.2 Object movement between ASOS and PSOS - Loading and
Unloading

Objects may have to be transferred through network links; such objects
must be flatted. Even if both spaces are located on the same computer,
objects have to be flattened in order to be saved their passive form in the
non-volatile storage in the PSOS. The structure of a flatted object
depends on the object itself. The only thing to constrain is the basic unit,
called slot, of the structure (like bytes in Unix). All information is
transferred as a stream of slots with IOPs (see section 3.5.3). The PSOS
manager recognises the IOPs, and it can store and retrieve objects by their
IOPs.

As mentioned earlier, a private object space and a connected ASOS may
use the same object identifiers. IOPs are used for space independent names
in flatted objects. Identifier conversion is required when an object is
loaded into ASOS and when objects are sent to PSOS. Both conversions
are done by ASOS managers. Except for string oriented names which are
supplied as one of an object's attributes, corresponding AIDs must be
created and saved into a mapping table. These AIDs may change the next
time an object is loaded. However, some AIDs are coupled with
corresponding IOPs, and cannot be changed. The reason is that clients
must know at least AIDs to request services at the beginning. Reserved
AIDs are related with these IOPs. These <AID, IOP> pairs are used by a
binder service to find the location of a service - the binder understands the
reserved AIDs. Such tables are stored into PSOSs to prevent loss of the
table due to failures. When objects are sent to PSOSs, new objects may be
included. These new objects do not have IOPs; the IOP generator of the
service, which controls the naming scheme of the service, must be used to
create IOPs for these objects.

When a service starts, the entire service is usually not loaded into an
ASOS. Each service has an attribute called initial groups. These groups are
Objects are related with each other, and a group of related objects rather than an individual object is required to perform a specific function. A service consists of groups of objects to provide certain functionalities. When a service is created, objects belonging to the service are categorised into groups according to their relationship with each other in the service. When an object in a service is required from an ASOS, the group of objects to which the requested object belongs is loaded. A group of objects is treated as a basic unit of transfer between an ASOS and an FSOS.
Chapter IV
Cache Coherence Protocol

In this chapter, we describe a cache coherence scheme, which maintains consistency between replicas. First of all, we describe several alternative replication schemes, and justify the one we have chosen. Then we describe the cache coherence protocol and its relationship with concurrency control protocols which are used to maintain consistency in context of transactions, followed by comparison with other concurrency control schemes. Then we explain transactions, version control, and a commitment algorithm used in our protocol. Finally the characteristics and advantages of the new protocol are emphasised.

4.1 Replication and Consistency

A distributed system consists of a set of services which may be shared by users. A service can be replicated to increase its availability and reliability. When a replica of the service is not available due to a node failure, other replicas are readily available. Replication reduces the effect of a system failure and it increases the reliability of the system. Availability can be enhanced further by locating replicas near users.

We refer to a set of objects providing certain functionalities for clients as a service and a machine that executes services and provides storage facility for services is called a server. A storage is called permanent if its contents are recoverable after system failures; otherwise it is called temporary. A permanent storage is called a repository and a temporary storage is called a cache. If the storage stores exclusively objects then it is called object storage, and if a server runs services which consist of objects and provides object storage, it is called an object server. In our object space model, ASOSs are used as caches and PSOSs as repositories. ASOSs provide a running
environment for services and also execute them and PSOSs provide permanent storage facilities for services.

A service can be replicated in two ways: by cache replication and by repository replication. Cache replication means a service is replicated into temporary storage to increase its performance. After the final reference to a replica, it can be removed from the cache. Repository replication means a service is replicated into a permanent storage to increase performance and robustness. The repository service cannot be removed from the storage even after the final access of a process. Both forms of replication require that the consistent state of the service is maintained even if repository replication must satisfy stronger constraints. We can consider four types environment:

1) a single repository (no replication),
2) cache replication with a single repository,
3) repository replication only, and
4) cache replication and repository replication.

1. A single repository (no replication): a service is located only in a repository and users must send all their requests to the service in the repository. Therefore, the service must provide concurrency control for concurrent access of the service and a running environment for objects in an object-oriented system. This scheme has a unique advantage - all services are centralised. Concurrency control and deadlock detection can be very efficient. Because there is only one copy in the system, pessimistic concurrency control becomes very efficient, and gathering information to detect deadlock requires no network communication.

However, there are several disadvantages in this configuration:

- Repositories are very complex in comparison with the alternatives because they save objects and also have to run objects. Therefore, such a repository must also be a server because it is not just a storage for services.
All requests must be sent to the services in repositories. Communication between two computers is an expensive operation. To send a message, objects related with the message require pre-processing to pass through the computer network and post-processing to be recognised by the services in the repositories. This takes a considerable time.

Repositories can be heavily loaded, which reduces the processing speed of a request and users' processes depends on the speed of repositories.

The system is not very robust. Even though the services in repositories can service node failures by using a recovery scheme, the service cannot be accessible before it has recovered from a failure.

2. Cache replication with a single repository: a service is represented by a single replica in a repository together with any number of replicas in caches. Because no service is stored at two different repositories, there is no consistency problem on the repository layer. The main advantage of this configuration is that the users requests are distributed to local replicas of the service instead of being sent to the service in the repository, which increases the availability of the service. However, maintaining the consistent state between replicas can be very expensive. Any change made by a process using a replica must be reflected into other replicas. To avoid this problem, many systems, such as Sprite [Nelson 88], Apollo DOMAIN [Levine 87], disable caches and use centralised concurrency control for concurrently used services.

3. Repository replication: a service is replicated into several repositories. Users can use any replicas of the service. However this alternative causes a consistency problem as in cache replication. To solve the problem effectively, several repository replication methods, such as majority-consensus [Thomas 79], weighted-voting [Gifford 79], Quorum-consensus [Herlihy 86] have been proposed. These schemes enhance availability and reliability of services. However, maintaining consistency between replicas becomes more complex than with a single repository because replicas must participate in concurrency control. Repositories must be object servers because they run services and store them.
4. **Cache replication and repository replication**: A service is replicated into caches and repositories. This maximizes the benefits of replication. However, it increases the complexity of the consistency problem because replica consistency must be maintained in caches as well as in repositories. This problem must be solved without a heavy overhead. In this replication method, caches can execute the service and repositories can provide storage for services.

We propose a cache coherence protocol which can be used to solve the consistency problem with cache and repository replication. The protocol provides a strong consistency in the repositories but a weak consistency in caches. In other words, the states of services in repositories are always consistent but they may not be always consistent in caches. To relax the consistency requirement in caches, the protocol includes concurrency control and recovery facilities.

Services are replicated at several repositories. A cache may be loaded from any single repository. Ideally, updates are propagated to all of the replicas. Although we are not dealing with partitions, the repository replicas are not always available to receive updates. In practice one of the replication methods discussed in Chapter 2 may be used, for example Quorum Consensus.

In this chapter we make the assumption that updates are sent to all available repository replicas. However we do not discuss issues relating to their recovery until Chapter 6. The partition of repositories will be discussed under future work in chapter 8.

4.2 **Cache Coherence Protocol**

In a tightly coupled multiprocessor architecture, consistency is maintained via synchronised invalidation with hardware support and by reloading invalidated data from the shared main memory.

In distributed systems, this method has a major drawback: network communication between processes is slow and unreliable. Therefore, synchronised communication is unrealistic. We propose a cache coherence protocol which can be used in distributed systems without a
heavy communication overhead. As in tightly-coupled architectures, we assume all transactions use the most up-to-date states of objects. Cache update messages are sent out each time a transaction commits to all replicas of objects affected by the transaction, and all replicas in caches are updated immediately or after a delay, rather than invalidating and then reloading.

However, due to communication delay or message loss, replicas can be out-dated and transactions may read the out-dated information. As a result, these transactions may cause inconsistent states. If they are inconsistent, they will be aborted when they attempt to commit.

Some transactions which read out-dated states of objects, can be saved from abortion by using local information in objects. Updating a cache can be deferred depending on the situation in the cache. For example, when replicas of objects in a cache are used by a read-only transaction, updating of replicas in the cache is deferred until the read-only transaction commits.

Let us suppose that each transaction uses its own isolated replica of a service and when a transaction commits, it creates a new version of the service in a PSOS. If a transaction starts with the most recently updated version of a service and completes before the version is updated by another concurrent transaction, it correctly transforms a consistent state into another consistent state - the transaction needs no validation.

Interference by other concurrent transactions is seen when the replica of a service used by a transaction suddenly becomes an out-dated version. Such interference can happen whenever a transaction commits. This means that as long as replicas are up-to-date, transactions using these replicas do not require any validation because they operate on the most recent version of a service all the time. There are several problems related to updating replicas across a computer network:

1. Sending an update message to other replicas in ASOSs takes propagation time.
2. Such update messages may be lost during propagation.
3 The receiving sequence of these messages may not be the same as their sending sequence.

We accept inconsistency caused by delay due to propagation and to transactions' computation time, and pay the price by aborting transactions which have used out-dated version of replicas of the service.

Transactions can be executed concurrently in ASOSs. When a transaction commits, the a new version of affected objects are saved in PSOS replicas and at the same time, the changes are propagated to all replicas of these objects. This is achieved by multicast communication. If an active transaction has read an out-dated state of an object, then the transaction is aborted. The restored state of the replica used by the aborted transaction has the most up-to-date version rather than the version used by the aborted transaction when it started. When the transaction restarts, the updated version of objects is already in the ASOS.

The second and the third problems are solved by using version numbers for objects in our system. Object replicas in ASOSs have version numbers. The initial values of version numbers of these replicas are supplied when they are loaded into an ASOS from a PSOS manager. Whenever an object is updated at the end of a transaction, the version number of the affected replica is increased by one. The new version number is saved with the new state of the object in the PSOS replicas. Any consecutive updating messages on an object must have consecutive version numbers.

If two version numbers are the same then the version having smaller replica number is considered older than the other. The replica number of an object is used to distinguish simultaneous increment of version numbers by two different replicas. The receiving sequence of such update messages are not important. If an object receives an update message, it checks its version number. If the difference between version number of the object and the update message is greater than one, some update messages are missing. However, objects can still carry out updates because the missing states of the object are overwritten by this update.
Update messages are sent as part of the two-phase commitment protocol for a transaction in an ASOS. The only required response is from corresponding objects in the PSOS or a sub-set of replicated objects if the service is replicated into several PSOSs. The chosen sub-set of PSOS replicas depends on the method of repository replication in use.

Communication between objects in the ASOS and in the PSOS uses multicast. As seen in Figure 4.1, one-way communication is used between cache replicas - the receiving replicas do not send any reply. Cache replicas receive the same information as replicas of the service in PSOSs, so they can update their state without contacting replicas of the service in PSOSs. Because a PSOS supports ASOSs with its storage, the PSOS is not concerned with consistency in caches, but updates states of objects if the update requests are valid, i.e., the requesting object must be the most up-to-date one.
When a transaction tries to commit, each service in an ASOS used by the transaction sends a prepare-commit message to the service in the PSOS replicas. The prepare-commit message contains update information and is sent via multicast. The PSOS managers either accept the prepare-commit or ask the transaction to re-commit. If the prepare-commit message is accepted, the PSOS manager send a 'can-commit' message via unicast.

A PSOS manager may send a 're-commit' message if the committing transaction may cause conflicts with either already committed transactions or other concurrently committing transactions. Note than PSOS managers do not reject prepare-commit messages. Abortion of a transaction (actual rejection of commitment) is decided by replicas in ASOSs used by the transaction. (see section 4.6.2)

4.3 Concurrency Control

As mentioned in the previous section, transactions are supposed to read the most up-to-date state of replicas in ASOSs. If they do not satisfy this requirement, they are aborted unless violations can be resolved. Because we know the total sequence of committed transactions and version information for all objects and replicas, we can use this information to rescue transactions. In the rest of this section we will explain how to resolve the violation of the requirement to avoid unnecessary abortions of transactions.

Operations requested by concurrent transactions may cause objects to have inconsistent states. Therefore conflicting operations should not be permitted to be executed. However, as cache replicas have only local information, conflicting operations in different replicas may be executed concurrently. These conflicts are detected when update information is received from another replica when a transaction commits. Unless such conflicts are resolvable, transactions that requested conflicting operations must be aborted to prevent objects from having inconsistent states.

When all operations are divided into read and write operation categories, four possible conflicts can be considered. In each case we are comparing an operation in a later active transaction with an earlier operation of a transaction that has just committed.
Read operations with other read operations are not in conflict because a read operation does not cause a state change.

Conflicts between pairs of write operations can be resolved by considering the effect of one write operation to be over-written by the other one.

In our protocol, states created by the committed transaction are always considered as the overwritten states by write operations of active transactions in the receiving replicas. If an active transaction is aborted, then the objects affected by the aborted transaction restore their states according to the received state rather than the original state with which the transaction started. Note that inter-replica communication occurs only when a transaction commits: conflicts are detected between the committed transaction and active transactions, but not between active and active transactions.

Write (active) and read (committed) operations are also not in conflict. This is because the committed transaction read from earlier versions of the objects involved.

Read (active) and write (committed) operations can be in conflict in our protocol. There are two cases:

1. The read operation of the active transaction occurred before the other transaction committed. In this case there is a conflict. In this case, the active transaction had read an out-dated state of a replica before the effect of a write operation of a committed transaction arrived in the ASOS due to concurrent execution of transactions and communication delay. The active transaction is aborted unless the transaction is a read-only one.

Because objects are capable of supplying certain versions of their state even after updating, the read-only transaction can continue its operations; the transaction sees the version with which it started until it completes.

2. The read operation of the active transaction occurred after the other transaction committed and they are not in conflict. Active transactions read states of replicas from the newly updated version,
unlike in optimistic concurrency control. Active transactions can access more than one version of the state of objects during their lifetime. We update replicas as soon as possible to avoid this kind of conflict.

4.3.1. Local concurrency control

So far we have considered concurrent transactions accessing different replicas. Concurrent transactions within a replica are permitted. A service may have multiple active copies, each of which uses a local conflict-based concurrency control protocol to maintain consistency. Replicas of a service can increase its availability and the use of a local conflict-based protocol can reduce the number of replicas required in the system. Using a replica serially or concurrently is hidden from other replicas because they only see the atomic-updated series of the object state rather than individual changes in the replica.

However a local concurrency control protocol does not have exclusive power over an object, because it maintains a serialisable sequence within a replica and it has no ability to serialise transactions in other replicas. However all objects in the system observe the transaction ordering generated by the cache coherence protocol. In other words, two protocols: the cache coherence protocol and a local concurrency control protocol are used to serialise transactions.

The cache coherence protocol will, if necessary, invalidate local concurrency control. Because a service in our system is an almost self-contained unit, all objects in a service use the same concurrency control scheme. This reduces the overhead due to the usage of different concurrency control protocols. Figure 4.2 shows an example in which four transactions, \( T_1 \)- \( T_4 \) uses two cache replicas, \( O_{r1} \) and \( O_{r2} \), of an object, \( O \). \( T_1 \) and \( T_2 \) use \( O_{r1} \), and \( T_3 \) and \( T_4 \) use \( O_{r2} \). Both \( O_{r1} \) and \( O_{r2} \) use two-phase locking to control concurrency. \( T_1 \) has set a read lock on \( O_{r1} \) and \( T_2 \) is waiting to write it. \( T_3 \) has set a write lock on \( O_{r2} \) and \( T_4 \) is waiting to read it. No transaction is committed yet.
We can consider six possible commitment sequences: \(<T_1, T_2, T_3, T_4>,\)
\(<T_1, T_3, T_2, T_4>,\)\(<T_1, T_3, T_4, T_2>,\)\(<T_3, T_4, T_1, T_2>,\)\(<T_3, T_1, T_4, T_2>,\) and
\(<T_3, T_1, T_2, T_4>\). Other sequences cannot occur due to locks. We assume
that when a transaction is committed, preceding transactions in a sequence
are committed and the following transactions are active. In \(<T_1, T_2, T_3, T_4>\),
no transaction is aborted. Between \(T_2\) and \(T_3\), there are write-write
operations but they do not cause conflict. In \(<T_1, T_3, T_4, T_2>\), no
transaction is aborted because write-write conflict between \(T_3\) and \(T_2\) is
ignored - replica 1 is not aware the existence of \(T_4\). In \(<T_3, T_4, T_1, T_2>\) and
\(<T_3, T_1, T_4, T_2>\), \(T_1\) is aborted due to conflict between \(T_3\)'s write and \(T_1\)'s
read - note that the read lock is set before \(T_3\) is committed because initially
no transaction is committed and \(T_1\) had already set the read lock; it is read-
write conflict rather than write-read conflict. \(T_2\) can commit because its
conflict with \(T_3\) is write-write conflict because \(T_3\) committed before \(T_2\)
commits and \(T_2\) does not know the existence of \(T_4\) because \(T_4\) had only
read operation, therefore \(T_4\) send no update message. In \(<T_3, T_1, T_2, T_4>\),
\(T_1\) is aborted as in \(<T_3, T_4, T_1, T_2>\) and \(<T_3, T_1, T_4, T_2>\). \(T_4\) will be aborted
if \(T_4\) set a read lock before the update information generated due to \(T_2\)'s
commitment arrive at replica 2 (read-write conflict); otherwise \(T_4\) can
complete has read-write conflict with \(T_3\) and \(T_4\) with \(T_2\) (write-read
conflict).

4.4 Comparison with Concurrency Control Protocols

Concurrency control schemes are categorised into optimistic and
pessimistic strategies and into single version or multiple version schemes.
An optimistic scheme assumes that conflicts between concurrent transactions will not affect the consistent state of the object and after a transaction completes, it validates the assumption, whilst pessimistic schemes assume that conflicts will violate the consistency constraints, so some conflicting transactions are blocked until the conflict is resolved or their effects are discarded. In a single version scheme, an object has only one state and as soon as it changes its state, it forgets its old version, meanwhile in multiversion schemes the object remembers its old versions and transactions may access them.

Because of the assumption that all transactions use the most recent versions of objects, ours can be seen as a kind of optimistic approach. In optimistic concurrency control protocols, conflicts happen between sets of write and read operations. Backward sequential validation uses a read set, \( R(T_i) \), of objects read by a committing transaction, \( T_i \), and write sets, \( W(T_i) \), of objects written by all committed transactions, \( \{ T_i \mid i \in 1..n \} \), which committed after the committing transaction started. Whenever any intersection, \( W(T_i) \cap R(T_j) \), of read and write sets of two transactions is not empty, the committing transaction is in conflict and it is aborted to resolve the conflict. In addition if two transactions try to commit concurrently, the intersection of their write sets, \( W(T_i) \cap W(T_j) \), must be empty; otherwise one of them must be aborted (see section 4.6).

Our cache coherence protocol is designed to reduce conflicts between these write sets of committed transactions and the read sets of active transactions. In ordinary optimistic concurrency control transactions are validated at the end of their execution. Therefore, they do read out-dated states of objects even when this can be avoided by reading from write sets of committed transactions. The cache coherence protocol supplies these write sets to active transactions as soon as possible at minimum cost.

The updating of objects can be seen as the validation of all active transactions against the committed transaction. Note that it is not validation of the committed transaction. This property provides two advantages:

- If transactions are already in conflict before updating, they will definitely fail the validation of optimistic concurrency control. By
aborting them as soon as possible, we can save computing power. It also increases concurrency by releasing its blocking against other transactions in the ASOS.

- It prevents write-read conflicts by supplying up-to-date state to transactions which access these states after updating. In addition the re-starting cost of aborted transactions is reduced by avoiding reloading of active services because the active services used by the aborted transaction already has the current values of objects.

Another property of the cache coherence protocol is that unlike ordinary optimistic concurrency control, it exploits the execution order of transactions. This property rescues some active transactions from abortion:

- Write-write conflict is ignored by considering write operations of active transactions as overwriting.

- If a read operation of a transaction is executed after a write operation of the same transaction on an object, the object is not included in the active transaction’s read set even if these operations are done before updating. In ordinary optimistic concurrency control, this transaction may be unnecessarily aborted if the object is the only one involved in conflict.

For read-only transactions, the cache coherence protocol provides outdated but consistent states of objects. A read-only transaction is ordered between the transaction which created the version with which the read-only transactions start and a transaction which creates the next version of the object. For example, a read-only transaction, T₂, started with a version which is created by transaction T₁. If T₃ creates a new version before T₂ finishes, then the committed transaction order is T₁, T₂, and T₃. If an object which is not loaded at an ASOS when the new version is created is required by T₂, the object, which has the new version, is loaded from a PSOS with its out-dated version. This property is similar to that of multiversion concurrency control.
4.5 Transactions and Version Control

A transaction in our model is an object which guarantees that the entire requirements requested by a client are executed or are not executed at all. Each transaction has an identifier when it is created from its class object, and it sends the identifier with all requests. The identifier is used by services and PSOS managers to control requests from the transaction. Because a transaction uses services directly and indirectly via objects, the identifier must be unique. Such a identifier can be generated as suggested by Lamport[78].

There are two kinds of version control in our system; passive and active version control. Passive version control is done in PSOSs. PSOSs store committed states of objects. Objects in a service are grouped together and each group has its version number as one of its attributes. Whenever objects in the group are modified, the version number is increased by one. The increment is done by a PSOS replica when it accepts a commitment. The version number of a group, therefore, represents the current version of that group. The version number is used to verify our assumption that the committing transaction used the most recent version of an object. If the version numbers of the replica it used and of the object stored in the PSOS are the same then the committing transaction meets the requirement; otherwise the transaction has to re-commit (see section 4.6). If the service is replicated among PSOSs, the version number of the replica must be the same as the all of the PSOSs.

Active version control updates replicas in ASOSs. Several replicas of a service can run concurrently in different ASOSs. The most recent version in an ASOS is called the current version of the service in the ASOS. Whenever the state of active objects is changed by a committed transaction, replicated active objects in other ASOSs also update their states according to the changes made by the committing transaction. Replicas in ASOSs may be outdated due to communication delay or message loss. Communication delay make replicas temporarily outdated, but eventually replicas can update. When an object in a ASOS is outdated due to message loss, it can be updated by means of version information
sent by a PSOS (see section 4.6). The version information is stored in the history file (see chapter 6).

By using the version information, we can avoid re-loading unaffected objects in ASOSs from the PSOS. This information can be obtainable when a transaction tries to commit. Because the version used by the transaction is out-dated, the PSOS rejects its commitment and supplies the necessary information for the replica to be updated in the same message - no extra communication is required.

4.6 The Commitment Algorithm

In this section, we describe a commitment algorithm that reflects state changes of objects in ASOSs, made by a transaction, to their corresponding objects in PSOS replicas consistently. Then, we will suggest some improvements in the algorithm. The commitment uses two phases - prepare-commit and commit phase. During the prepare-commit phase, objects store presumed update state to PSOS replicas and receive can-commit message from PSOSs. During the commit phase, objects send commit messages to PSOSs and then they update their states after receiving committed messages from PSOSs.

4.6.1 Two-phase Commitment Protocol

When a transaction tries to commit, the transaction object sends prepare-commit messages to objects in ASOSs accessed directly by the transaction. Objects that receive the prepare-commit message propagate the message to other objects used by the object.

Each object, then, performs local validation based on version information. It sends its version number and the new state if any new state is created by the committing transaction. The states are saved in PSOS if the two version numbers in the ASOS and in the PSOS match. To avoid generating unnecessary messages, only one message is sent to the PSOS with a set of group identifiers, their version numbers, and states - messages sent by objects in the service are gathered and duplicate messages to the same group are removed.
If the version numbers of each group are the same as the current active groups, the PSOS replica replies with a can-commit message; otherwise, it sends re-commit messages with information. If the object receives a can-commit message, it replies with a can-commit message to the sender of the prepare-commit message.

If objects receive re-commit messages, they update their own state if it is out-dated. If any conflict is found in an object at this stage, the object rejects the commitment of the transaction by sending an 'abort' message to the transaction object. Otherwise they send out a prepare-commit message again (see also section 4.6.2). A reply message also contains the service and group identifiers of the object and the address of the PSOS which the object contacted. The information is used at the second phase.

After a transaction object receives all the replies, it starts the second phase. If all replies are can-commit, it sends a commit message. Otherwise, it sends an abort message. Both messages are sent to all PSOSs which are included in reply messages of prepare-commit messages. In other words, multiservice commitment is required to commit the transaction (see section 4.6.3). After receiving the commit (or abort) message, the states of affected services in each PSOS are updated (or discarded) by the PSOS managers. The manager sends a committed message to the transaction. Any failure situation at either ASOS or PSOS is explained at section 4.6.3.

4.6.2 Concurrent Commitment

When more than one transaction is trying to commit, the PSOS manager at a replica takes the first prepare-commit message and others are tested for whether they can commit concurrently or must wait until the first transaction finishes the commitment. PSOS managers permit concurrent commitment if there is no conflict between two write sets, sent by replicated objects on behalf of different transactions. The intersection is calculated with object groups rather individual objects. Because objects are encapsulated, the PSOS does not know objects' structure. If the intersection of these write sets (object groups) is empty, then they can commit concurrently. In other word, the PSOS manager can reply to more than one prepare-commitment in such a case. If the intersection is not empty, the PSOS sends a 're-commit' message and sends with it the
conflict write set. The transaction has three chances to survive: 1) there is no conflict indeed, 2) the conflicting transaction is aborted or 3) the conflict can be solved after the conflicting transaction is committed if the transaction has write operations before any read operation on conflicting objects; new states of conflicting objects are created by overwriting. Therefore, PSOS managers send reply messages to all objects that send prepare-commit messages to them.

Each reply message contains information as to whether they can commit or have to re-commit. Then, each object decides its next action. If an object receives the can-commit message from its PSOS service, then it can continue. If it receives a re-commit message, it can choose either re-commit or abort the transaction after examining the write set sent by the PSOS. If a transaction is not in conflict or in conflict but resolvable then it re-commits; otherwise it aborts. If the transaction is not in conflict then it sends another prepare-commit message to the PSOS. If the transaction is in conflict but can resolve the conflict after the commitment of the transaction with which the transaction is in conflict. In the latter case, the object asks the sender of the prepare-commit message to release prepare-commit blocking to avoid commitment deadlock [Ghertal 85].

Commitment deadlock occurs when more than one repository (PSOSs) involved in concurrent commitment of transactions. As only one transaction at a time can use a particular object, only one transaction at a time can perform a pre-commit on an object. When a transaction is performing a pre-commit on an object we say it holds a prepare-commit blocking on that object.

If a transaction T₁ committing concurrently with a transaction, T₂, holds a prepare-commit blocking on an object, O₁, and waits on an object, O₂, and, T₂ hold a blocking on O₂, and waits on O₁, then both transactions cannot ever commit. In our protocol, a transaction having smaller transaction identifier releases its blocking to avoid this problem.

For example, suppose that two transactions, T₁ and T₂, try to commit concurrently and both used two services, S₁ and S₂. The commitment request of T₁ arrives before T₂ at S₁ and T₂ before T₁ at S₂. For S₁, T₁’s commitment is granted and T₂ is waiting. For S₂, T₂’s commitment is
granted and T1 is waiting. As a result, both cannot commit; they cause a commitment deadlock. Because T1 is less than T2, T1 cannot hold prepare-commit at S1, so T1 releases its prepare-commit at S1. After T2’s commitment, it tries to commit again - it sends prepare-commit message to objects to find out whether it can commit. If it chooses to abort the transaction, then it must notify the sender of the prepare-commit message and the PSOS service to release prepare-commit blocking for other transactions.

Figure 4.3 Message flow for a committing transaction

Figure 4.3 shows the message flow of a committing transaction. A transaction sends a prepare-commit to object A accessed directly by the transaction. The prepare-commit message includes the location of the transaction in order that objects accessed indirectly by the transaction can communicate with the transaction directly. Then, object A forwards the message to object B, then object B forwards it to object C in another service
2. Then object A and object B send a prepare-commit message to service 1 in a PSOS and object C sends it to service 2 in a PSOS. Then each service in the PSOS replies to the prepare-commit message to the corresponding object; service 1 sends can-commits to object A and object B and service 2 sends a can-commit message to object C. Then object C replies with a can-commit message to object C and then object B replies to object A. Finally object A replies to the transaction. Then the transaction starts the second phase. It sends a commit message to objects in PSOSs. At this time the transaction sends the commit message directly to PSOSs because now the transaction knows all involved PSOSs and their address. Then services in the PSOS sends committed messages to object A, B, and C.

4.6.3 Multiservice Commitment

A transaction may use more than one service and they may be located at different nodes. If two services in Figure 4.3 are located at different ASOSs, then the transaction requires multiservice commitment. Our situation is a little different from ordinary multiserver commitment [Gray 79] [Lampson 81] [Coulouris 88].

The role of a coordinator which coordinates the commitment of transactions between several servers is done by a committing transaction which is located in an ASOS. In addition each object affected by the committing transaction uses multicast to propagate update information, and communication is done between objects in ASOS and in PSOS. Furthermore the transaction has no idea in advance which services it will use. Such information is only available at the end of the first phase, i.e., after it receives all replies of its prepare-commit messages from objects.

If the ASOS containing the coordinator fails after the PSOS replicas have prepared to commit but before the transaction has committed, the situation is not recoverable.

To prevent this situation from occurring, the transaction chooses the PSOS replica of an object it used directly as a kind of a backup coordinator which completes the commitment phase in the case of failure. It also performs the second phase even when there is no failure.
The chosen PSOS replica does not know which services are involved before it receives the information from the transaction after the first phase. Therefore, it must broadcast to find out whether the transaction has failed if it does not have the required information after a time-out: If the transaction does not respond, the chosen service aborts the transaction by sending a broadcast message which can be received by all replicas involved.

At the beginning of the second phase, the transaction object sends a commit message which includes the list of services used by the transaction to the backup coordinator via a point-to-point communication and waits for the reply. The backup coordinator multicasts a committed message to all PSOS and ASOS replicas, which include the transaction object.

All other PSOS replicas involved in the transaction send replies to the backup coordinator (not the transaction object). After receiving committed messages from all PSOS replicas involved in the commitment of the transaction, the backup coordinator finishes its role of the coordinator. If any PSOS replica does not send a reply, the backup coordinator contacts it. If the PSOS replica is in a failure, the backup coordinator cannot complete its task until the failed replica recovers.

We can consider four types of possible failures:

- an ASOS fails during the first phase,
- a PSOS fails during the first phase,
- an ASOS fails during the second phase, and
- a PSOS fails during the second phase.

An ASOS fails during the first phase: a transaction cannot pass the first phase because at least one object cannot send the prepare-commit reply. The transaction will be aborted by itself.

The ASOS containing the transaction object may fail after sending prepare-commit messages but before it has collected all the replies. This case is handled by the backup coordinator as described earlier.
A PSOS fails during the first phase: objects loaded from the failed PSOS cannot receive any reply from the PSOS, so these objects refuse the commitment of the transaction. The transaction will be aborted by the transaction manager. In the case that the PSOS is replicated, this problem is less severe.

An ASOS fails during the second phase: an ASOS failure in this phase does not affect the transaction because the commit message does not go through objects. The second phase of the protocol is performed by the backup coordinator.

A PSOS fails during the second phase: when a backup coordinator fails, the service cannot commit until it recovers.

But the other PSOS replicas have already prepared to commit and will have to wait until the backup coordinator recovers. After recovering, backup coordinator sends a committed message according the information given by the transaction before it crashed.

Summary of the commitment procedure

Phase One

1. Before it starts the first phase, a transaction object chooses a PSOS to become the backup coordinator. The transaction object, then sends the transaction identifier to the backup coordinator. After this, the transaction object issues prepare-commit messages to objects accessed directly.

2. Objects forward the message to other objects accessed via them. They send prepare-commit messages with new states (if any) to PSOS managers which maintain their passive images.

3. PSOS managers reply to prepare-commit message to objects that sent the messages. If the version of objects is the same as theirs and there is no concurrent commitment request, then they send can-commit messages. If the version of objects is out-dated or there is any concurrent committing transaction, then they send re-commit
messages. The re-commit message includes update information with it.

4. After receiving the replies, objects reply to the transaction object with their service and group identifiers and whether the transaction can-commit, has to re-commit or must be aborted.

5. After collecting all the replies, the transaction starts the second phase if all replies are can-commit. If some of them have to re-commit, the transaction releases its prepare-commit blocking at PSOSs to avoid commitment deadlocks and then sends another prepare-commit message to objects. If any object replies with abort, then the transaction multicasts an 'abort' message so that the previous prepare-message can be removed from PSOSs and replicas in other ASOSs can ignore the prepare-commit message.

Phase Two

6. The transaction sends a commit message to the service which was chosen as the backup coordinator of the transaction via point-to-point communication. The commit message includes a list of all the services used by the transaction and their addresses.

7. The backup coordinator sends a committed message via multicast to all ASOS and PSOS replicas. After receiving the committed message, other replicas update their states and send committed messages to the backup coordinator. After receiving all committed messages from other services in PSOS, the coordinator service removes information related with the transaction. The coordinator must receive the commit message from the transaction before timeout; otherwise it broadcasts that the transaction cannot commit.
4.7 Advantages of the Cache coherence Protocol

Our cache coherence protocol aims to update caches as soon as possible, as in a tightly-coupled multiprocessor architecture. The protocol provides several advantages:

- The protocol minimises the overhead needed to maintain replicated objects in caches. It uses messages generated by the two phase commitment protocol to supply the update information to caches. This dual use of these message reduces the amount of communication for cache update.

- The protocol replaces concurrency control for concurrent transactions. Because the protocol is used by objects in caches, the task of maintaining consistency is moved into caches. In a non-replicated server, all concurrency control could be centralised, but it is distributed in our design.

- It reduces the abortion rate of transactions relative to that of an optimistic concurrency control protocol. By supplying update information as soon as possible, it reduces the chance of write-read conflicts. In addition, it considers operation orders in a replica of an object in order to resolve additional write-read and write-write conflicts which may save some transactions that are normally aborted in an optimistic concurrency control. For read-only transactions, the protocol looks like a multiversion concurrency control. It supplies the version with which a read-only transaction started until the transaction finishes even if the version becomes out-dated.

- The protocol permits the sharing of a replica by more than one transaction. Local transactions running concurrently are controlled by a conflict-based concurrency control such as two-phase locking or timestamping. These protocols observe the transaction order generated by the cache coherence protocol. In other words, local concurrency control is valid only within a replica. If there is only one active replica, then its behaviour is the same as that of an ordinary conflict-based concurrency control protocol.
Recovery and version information is combined into a single file, called the history file to utilise common storage space (see chapter 6).
Chapter V
Correctness of Cache Coherence Protocol

In this chapter, the correctness of our cache coherence protocol is argued. After defining sharable and exclusive states of an object, we will give some definitions relating to objects, operations and transactions in an ASOS. Then the correctness of the cache coherence protocol is shown, first without local concurrency control and then with local concurrency control. The last section deals with user defined concurrency control objects.

5.1 Sharable and Exclusive State

Entities in an object-oriented environment are not the same as conventional data items. Conventional data items are accessed by standard operations: read and write, whereas objects are accessible only with operations defined in their behaviour specification. An object may have either a simple structure or a composite structure. Examples of composite objects are arrays, sets, etc., whose state consists of several objects. Controlling concurrent access to an entire object may result in an unnecessary reduction of concurrency. Transactions may share subsets of the objects comprising a composite object without causing inconsistent state. As long as the intersection of the subsets accessed by concurrent transactions is empty, transactions do not cause inconsistent state and they should be permitted to access them concurrently.

Figure 5.1 shows an example of such a case. Items in Figure 5.1(a) represent the coordinates of two points. They can be accessed by ordinary read and write operations, and they can be shared between two transactions, i.e. different transactions use different items. In Figure 5.1 (b), a rectangle
object has two points representing its origin and corner. Two points can be accessed by operations defined in the behaviour specification of the rectangle object. When one transaction uses POINT 1 and another uses POINT 2, there is no conflict. Such sharing should be permitted to increase concurrency.

<table>
<thead>
<tr>
<th>ITEM 1</th>
<th>ITEM 2</th>
<th>ITEM 3</th>
<th>ITEM 4</th>
</tr>
</thead>
</table>

a) the structure of conventional data items

<table>
<thead>
<tr>
<th>OBJECT X1</th>
<th>OBJECT Y1</th>
<th>OBJECT X2</th>
<th>OBJECT Y2</th>
</tr>
</thead>
<tbody>
<tr>
<td>POINT 1</td>
<td>POINT 2</td>
<td>A RECTANGLE</td>
<td></td>
</tr>
</tbody>
</table>

b) the structure of a complex object

Figure 5.1 The structure of conventional data items and a complex object

Sharable and exclusive states of an object are introduced to cope with this problem. An object becomes exclusive if it is replaced by another object; otherwise it is sharable. The granularity of concurrency control depends on how to define the replacement. For example, changing an element of an array could be considered as replacement with another array which has the same state as that of the replaced array except the changed element. To reduce concurrency control overhead, the granularity must be coarse. To give maximum concurrency, the granularity must be as fine as possible.

For instance, concurrency control can be applied to an entire array to reduce concurrency control overhead or to each element of the array to increase concurrency. We use the term composite object to refer to a set of interconnected objects. If partially changing the state of a composite object is regarded as a replacement, such a composite object is called a user-defined concurrency control object. By using the semantics of applications, a user-defined concurrency control object may increase the efficiency of the
whole system or reduce overhead by avoiding too fine grain concurrency control.

5.2 The Correctness of the Cache Coherence Protocol

In this section, we will first give the basic definitions needed in our proof. Then we describe how to construct a directed graph for the cache coherence protocol only, and then finally for the cache coherence protocol with local concurrency control.

5.2.1 Basic Definitions

Each object in an ASOS does its own version control, and it decides whether a transaction can commit with respect to itself. A transaction can be committed only if all objects accessed by the transaction directly or indirectly agree to its commitment. An ASOS may also provide its own local concurrency control.

Definition 1 An active object O has a replica number, rn(O), a version list, vl(O), and a set, Objstate(O), of objects, which compose the state of the object. These objects are called the state-objects of the object. If the Objstate(O) of an object is empty, it is called a simple object; otherwise a composite object.

The replica number is a number assigned by a PSOS manager which stores the passive service, to distinguish replicas. The version list keeps all old and current objects which represent the value of the object. For example, a rectangle object, Rect, has two points, Origin and Corner, in its Objstate(Rect). If Rect is replaced with another Rect object, its version list contains two objects, the replaced and the replacing objects. But if one of the coordinates, X, of Origin is replaced with another, then the version lists of Rect and Origin contain only the current object, and the version list of the replaced coordinate, X, contains two objects.

Note that all entities in object-oriented systems are objects. In a conventional database system, a replacing item has the same type as the replaced item. However, some dynamic binding systems such as Smalltalk permit replacement of an object with another having a different class; their structures may be different. Therefore objects in a version list may
belong to different classes. An element in Objstate(O) is also an object, therefore, it has a replica number and a list of versions, and a set of object states.

**Definition 2** An operation is *terminal* if it references an object without causing partial change of its state or if it replaces any composite object with another. Each terminal operation, op, has three attributes: a transaction identifier, tid(op), the type of action, act(op), and an entity. An act(op) ∈ \{R,W\} where R represents reference operations and W replacement operations. The entity consists of an object identifier, oid(op), and two version numbers, vlbef(op) and vlafth(op) that are indexes in the version list of the object. For an R type of operation, vlafth(op) is empty. □

**Definition 3** A transaction is a finite sequence of terminal operations, T=op1,op2,...,opn. Non-terminal operations are decomposed into sub-operations and are replaced by them until there are no more non-terminal operations in its sequence. □

A transaction guarantees that all changes it makes are either done completely or not done at all. An operation consists of a request and its reply message. A request message sent to an object may generate other messages to complete the original request. The terminal operations are not interleaved with other operations. In other words, terminal operations are atomic.

**Definition 4** A history, h, of transactions, T1,T2,...,Tn, is a permutation of terminal operations of these transactions and it preserves operation ordering within a transaction. The transaction history, hT, and the object history, ho, denote the subsequence of history h consisting of all terminal operations, involving a transaction T and an object O respectively. T(h) and Obj(h) are the set of transactions and objects involved in the history h, respectively. If there is no uncompleted nor aborted transaction in a history, the history is called a complete history. □

When we mention a history, we mean a complete history from now on unless specified otherwise.
Definition 5 A history $h$ is serial if each transaction in $T(h)$ is executed completely before the next one starts. □

There are $n!$ possible serial histories for $T(h)$ where $n$ is the number of transactions in $T(h)$ and $P$ is the permutation function. Any operation which does not belong to a transaction $T$ must be either before all operations of $T$ or after all operations of $T$ in a serial history. Any serial history conserves the consistent state of objects.

5.2.2 Cache Coherence Protocol

We describe how to construct a directed conflict graph for history $h$. First of all, each object constructs its own dependency graph based on its local information. If an object has state-objects then it combines its directed graph with directed graphs of state-objects. Finally, the directed graph for the given complete history is the combined graphs of all the objects affected by the transactions in $T(h)$.

By the Algorithm 1, each object $O$ constructs its own directed graph, $D(O)$, based on its local history, $h_O$. The set, $V(O)$, of vertices of such a directed graph is the set of transactions which have operations on the object and the set, $A(O)$, of arcs in the graph represents conflicts among transactions belonging to $V(O)$. There is an arc $(u,v)$ if a transaction $u$ must finish before a transaction $v$. The version information is used to create arcs in the directed graph of the object. The relation represented by the arcs is a partial order relation. A relation is partial order if it is an irreflexive, asymmetric, and transitive relation. A directed graph for a complete history $h$ can be constructed by Algorithm 2. The constructed directed graph is used to find a serialisable scheme for transactions involved in the history.
Algorithm 1 Let $h$ be a complete history. There is a directed graph, $D(O) = (V(O), A(O))$, for each object, $O$, in Obj(h).

A-I: For each object, $O$, such that Objstate($O$) = $\emptyset$,

For each version, $vl(i)$, in the version list of object, $O$,

the set of vertices for $D(O)$ is

$$V(O) = \bigcup V_{vl}(i)$$

where

$$V_{vl}(i) = \{ \text{tid}(op) \mid \text{vlbef}(op) = i \text{ and op} \in h_O \},$$

the set of arcs for $D(O)$ is

$$A(O) = A_1 \cup A_2$$

where

$$A_1 = \{ <T, \text{tid}(op)> \mid \text{act}(op) = W, T \in V_{vl}(\text{vlbef}(op)), \text{tid}(op), \text{and op} \in h_O \}$$  \hspace{1cm} --- \text{CASE I-I}$$

$$A_2 = \{ <\text{tid}(op), T > \mid \text{act}(op) = W, T \in V_{vl}(\text{vlaf}(op)), \text{tid}(op), \text{and op} \in h_O \}$$  \hspace{1cm} --- \text{CASE I-II}

A-II: For each object, $O$, such that Objstate($O$) $\neq \emptyset$,

For each version, $vl(i)$, in the version list of object, $O$,

Let $D_{state}(O_k) = (V_{state}(O_k), A_{state}(O_k))$ be a directed graph of each object, $O_k$, in the set of Objstate($O$).

the set of vertices for $D(O)$ is

$$V(O) = V_{state} \cup V_{vl}$$

where

$$V_{state} = \bigcup V_{state}(O_k) \text{ and } V_{vl} = \bigcup V_{vl}(i)$$

where

$$V_{vl}(i) = \{ \text{tid}(op) \mid \text{vlbef}(op) = i \text{ and op} \in h_O \},$$

the set of arcs for $D(O)$ is

$$A(O) = A_{state} \cup A_{vl}$$

where

$$A_{state} = \bigcup A_{state}(O_k) \text{ and } A_{vl} = A_1 \cup A_2$$

where

$$A_1 = \{ <T, \text{tid}(op)> \mid \text{act}(op) = W, T \in V_{vl}(\text{vlbef}(op)), \text{tid}(op), \text{and op} \in h_O \}$$  \hspace{1cm} --- \text{CASE II-I}$$

$$A_2 = \{ <\text{tid}(op), T > \mid \text{act}(op) = W, T \in V_{vl}(\text{vlaf}(op)), \text{tid}(op), \text{and op} \in h_O \}$$  \hspace{1cm} --- \text{CASE II-II} \Box
The first half, A-I, of the algorithm 1 describes how to construct directed graphs of objects whose object state is empty. Dependencies between transactions using an object can be traced through their operations. A read operation does not cause dependencies with another read operation; but a write operation causes dependencies with read and write operations. CASE I-I and CASE I-II describe such dependencies. Each write operation replaces an object with another. CASE I-I describes dependencies between read operations of T, that accessed the replaced object (old version: \( v_{\text{ref}} \)) and the write operation, act(op), that replaced it. CASE I-II describes dependencies between the write operation, that replaces an object, and read operations of T, that accessed the replacing object (new version: \( v_{\text{alt}} \)).

The second half, A-II, of the algorithm 1 describes how to construct directed graphs for complex objects. Because objects in \( \text{Obj}_{\text{state}}(O) \) of a complex object, O, represent the state(value) of the complex object, the dependencies graph of the complex object should include all dependencies in objects in \( \text{Obj}_{\text{state}}(O) \).

Algorithm 2 There is a directed graph, \( D(h) = (V(h), A(h)) \), for a complete history h.

Let \( D(O) = (V(O), A(O)) \) be a directed graph of an object, O, where \( O \in \text{Obj}(h) \). The set of vertices for \( D(h) \) is

\[
V(h) = \cup V(O),
\]

the set of arcs for \( D(h) \) is

\[
A(h) = \cup A(O). \square
\]

Figure 5.2 shows an example of a directed graph for the given history h. There is a composite object A containing a sub-objects B and C, and five transactions in a history. Note that transaction T4 replaced object B with another object having a different structure. Figure 5.2(b) show this replacement and the structure of the object A. Each operation executed by transactions is represented by a tuple. Each tuple contains a transaction identifier, the operation type and two version numbers. The first version number is the version that existed before the operation and the second
version number is the version created by the operation. The second version number will be empty if the operation type is read.

\[ h = \langle T_2, R, X, v_1, \_ \_ \_ \_ \_ \_ \_ \_ >, \langle T_1, R, X, v_1, \_ \_ \_ \_ \_ \_ \_ \_ >, \langle T_2, W, X, v_1, v_2 \rangle, \langle T_3, R, C, v_1, \_ \_ \_ \_ \_ \_ \_ \_ >, \langle T_3, W, Y, v_1, v_2 \rangle, \langle T_4, W, B, v_1, v_2 \rangle, \langle T_3, W, C, v_1, v_2 \rangle, \langle T_5, W, Z, v_1, v_2 \rangle \]

\[ h_X = \langle T_2, R, v_1, \_ \_ \_ \_ \_ \_ \_ \_ >, \langle T_1, R, v_1, \_ \_ \_ \_ \_ \_ \_ \_ >, \langle T_2, W, v_1, \_ \_ \_ \_ \_ \_ \_ \_ > \]
\[ h_Y = \langle T_3, W, v_1, v_2 \rangle \]
\[ h_B = \langle T_4, W, v_1, v_2 \rangle \]
\[ h_Z = \langle T_5, W, v_1, v_2 \rangle \]
\[ h_C = \langle T_3, R, v_1, \_ \_ \_ \_ \_ \_ \_ \_ >, \langle T_3, W, v_1, v_2 \rangle \]

Figure 5.2 (a) A history \( h \) and histories for objects in \( h \).

Figure 5.2 (b) an object diagram representing the history \( h \).
Figure 5.2 (c) Directed graphs created by the algorithm 1: CASE I-I

Figure 5.2 (d) the directed graph created by the algorithm 1: CASE II-I, II-II

Figure 5.2 An example of a constructed directed graph for a history h

Figure 5.2(a) gives a history h and object histories for each object involved in the history. At object X, transaction T₁ has a read-write dependency with T₂; the dependency, marked ₁₁ in figure 5.2(c), is represented by an arc <T₁,T₂> created by CASE I-I. The composite object B is replaced by the operation <T₄, W, B, v₁, v₂>. Several dependencies are created due to this operation. Firstly, there are read-write and write-write dependencies between T₁, T₂, and T₃, which read version 1 and T₄ which creates version 2. These dependencies create arcs, marked ₂₁, by CASE II-I in Figure 5.2(d). Secondly, there is a write-read dependency between T₄ and T₅. The dependency is represented by an arc <T₄,T₅>, marked ₂₂, in Figure 5.2(d). The directed graph for the history h is the same as the directed graph in Figure 5.2(d).

Definition 6 A history h₁ is equivalent to a history h₂ if D(h₁) is isomorphic with D(h₂). □
Definition 7 A history \( h \) is serialisable if there is a serial history \( h_S \) equivalent with \( h \). □

Let us consider three serial histories, ordered by \( <T_1,T_2,T_3,T_4,T_5> \), \( <T_1,T_3,T_2,T_4,T_5> \), and \( <T_3,T_1,T_2,T_4,T_5> \), for the history \( h \) in Figure 5.2(a):

\[
\begin{align*}
h_1 &= <T_1,R,X,v1_1>, <T_2,R,X,v1_1>, <T_2,W,X,v1_1,v2>, <T_3,R,C,v1_1>, \\
&\quad \quad \quad <T_3,W,Y,v1_2>, <T_3,W,C,v1_2>, <T_4,W,B,v1_2>, <T_5,W,Z,v1_2> \\
h_2 &= <T_1,R,X,v1_1>, <T_3,R,C,v1_1>, <T_3,W,Y,v1_2>, <T_4,W,B,v1_2>, <T_5,W,C,v1_2>, \\
&\quad \quad \quad <T_2,R,X,v1_2>, <T_2,W,X,v1_2>, <T_4,W,B,v1_2>, <T_5,W,Z,v1_2>, \\
&\quad \text{and} \\
h_3 &= <T_3,R,C,v1_1>, <T_3,W,Y,v1_2>, <T_3,W,C,v1_2>, <T_1,R,X,v1_1>, \\
&\quad \quad \quad <T_2,R,X,v1_2>, <T_2,W,X,v1_2>, <T_4,W,B,v1_2>, <T_5,W,Z,v1_2>.
\end{align*}
\]

These three histories have the same directed graphs as the history in the figure 3.4 (a). Therefore, the history \( h \) is serialisable.

Lemma 1 There exists a set, \( S_T \), of partial orderings between vertices if the set of vertices is not empty \( \) and \( D(h) \) is acyclic.

Proof. By definition of an acyclic graph there is a set of partial orderings among its vertices. □

Theorem 1 There is a serial history \( h_S \), of which a directed graph \( D_S(h_S) \) is equivalent with \( D(h) \) if \( D(h) \) is acyclic.

Proof. There is a partial order \( P_S \) in \( D(h) \) by lemma 1 and let \( h_S \) be a serial history which preserves \( P_S \). The two histories \( h \) and \( h_S \) contain the same operations. Therefore all vertices in \( D(h) \) exist in \( D_S(h_S) \) and vice-versa.

Let us show there are the same arcs in both \( D(h) \) and \( D_S(h_S) \). First, we show that an arc \((v,u)\) in \( D(h) \) also exists in \( D_S(h_S) \). An arc is created by four cases, CASE I-I, CASE I-II, CASE II-I, and CASE II-II : in \( h \) there is the operation, \( op \), causing the replacement of an object. All operations done by transactions which are reachable from the vertex \( , \text{tid}(op) \), in \( D(h) \) exist before \( \text{tid}(op) \) and all operations done by transactions which are reachable from \( \text{tid}(op) \) exist after \( \text{tid}(op) \) in \( D(h) \). Because \( D_S(h_S) \) preserves the given partial order \( P_S \), all operations belonging to transactions which are
reachable to (from) tid(op) exist before (after) the operation op. Arcs generated by CASE I-I, CASE I-II, CASE II-I, and CASE II-II exist in D5(hS).

Second, let us assume that an arc(u,v) exists in D5(hS) but not in D(h). Then it means either (a) transaction u has an operation, op, which replaces an object value with another in hS and transaction v has any read or write operations on the version which is created by op or (b) transaction v has an operation, op, in hS and transaction u has any read or write operation on the version which is replaced by op. The order of operations can be changed in the history of an object by re-ordering a history but an operation cannot disappear nor appear, therefore an operation exists before [(b) after] the replacement operation in hS but after [(b) before] in h. Such an operation creates an arc (v,u) in h. In other words, transaction u precedes transaction v in hS and transaction v precedes transaction u in h. Because hS must preserve the given partial order, P5, this is contradiction. □

**Theorem 2** The history h is serialisable if D(h) is acyclic.

**Proof.** By theorem 1, there is a equivalent serial history hS for h if D(h) is acyclic, and by definitions 6 and 7, a history h is serialisable. □

### 5.2.3 Cache Coherence Protocol With Local Concurrency Control

In this section, we explain how to construct a directed graph with local concurrency control. Two-phase locking (2PL) and timestamping (TS) are the best known protocols based on conflicts between operations. In 2PL, the position of a transaction in the sequence of transactions is decided when it completes and, in TS, it is determined when it starts. The former is called a dynamic protocol because the position of a transaction in the transaction sequence is unknown until it finishes and the latter static because the position of a transaction is known as soon as it starts.

When a service requires a highly protected environment or changes its state very frequently and is used by many users concurrently, a centralised conflict-based concurrent control is appropriate [Härder 84] [Bassiouni 88]. This case can be seen as a special case of the cache coherence protocol with
local concurrency control: there is a single copy of a service with local concurrency control.

Definition 8 A transaction $T_i$ precedes another transaction $T_j$, \( \text{precedes}(T_i, T_j) \), if $T_i$ commits before $T_j$ in a complete history $h$ or if $T_i$ has a higher timestamp than that of $T_j$. \( \square \)

The order of transactions controlled by a dynamic concurrency control protocol is determined when they commit, whereas it is determined when they start in a static concurrency control. A transaction controlled by 2PL has no timestamp. A transaction may be involved in more than one local concurrency control. Such a transaction must satisfy both orderings.

Algorithm 3 below constructs a directed graph which contains transaction dependencies between replicas and within a replica. The latter is created by local concurrency control protocol. Each replica of an object has a replica number, represented by $m$, in the algorithm.

Algorithm 3 Let $h$ be a complete history. There is a directed graph, $D_t(O) = (V_t(O), A_t(O))$, for each object, $O$, in Obj($h$).

A-I: For each object, $O$, such that $\text{Objstate}(O) = \emptyset$,

For each version, $v_t(i)$, in the version list of object, $O$,

the set of vertices for $D_t(O)$ is

$$V_t(O) = \cup V_v(t(i))$$

where

$$V_v(t(i)) = \{ \text{tid}(op) \mid v_{bef}(op) = i \text{ and } op \in h_O \},$$

the set of arcs for $D_t(O)$ is,

$$A_t(O) = A_1 \cup A_2 \cup A_3$$

where

$$A_1 = \{ <T, \text{tid}(op)> \mid \text{act}(op) = W, T \in T_v(t(v_{bef}(op))), T \neq \text{tid}(op), \text{ and } op \in h_O \} \quad \text{--- \ CASE I-1}$$

$$A_2 = \{ <\text{tid}(op), T> \mid \text{act}(op) = W, T \in T_v(t(v_{aft}(op))), T \neq \text{tid}(op), \text{ and } op \in h_O \} \quad \text{--- \ CASE I-2}$$

$$A_3 = \{ <\text{tid}(op_i), \text{tid}(op_j)> \mid \text{precedes}(\text{tid}(op_i), \text{tid}(op_j)),$$

$$\text{rn}(\text{oid}(op_i)) = \text{rn}(\text{oid}(op_j)), \text{ and }$$

$$\text{tid}(op_i), \text{tid}(op_j) \in V, \text{ op}_i, \text{ op}_j \in h_V \} \quad \text{--- \ CASE I-III}$$
A-II: For each object, O, such that Obj\text{state}(O) \neq \emptyset,

For each version, vl(i), in the version list of object, O,

Let D_{\text{state}}(O_k) = (V_{\text{state}}(O_k), A_{\text{state}}(O_k)) be a directed graph of each object, O_k, in the set of Obj_{\text{state}}(O).

the set of vertices for D_l(O) is

V_l(O) = V_{\text{state}} \cup V_{vl}

where

V_{\text{state}} = \cup V_{\text{state}}(O_k) and V_{vl} = \cup V_{vl}(i)

where

V_{vl}(i) = \{ \text{tid(op)} \mid \text{vlbef(op)} = i \text{ and op} \in h_O \}

the set of arcs for D_l(O) is

A_l(O) = A_{\text{state}} \cup A_{vl}

where

A_{\text{state}} = \cup A_{\text{state}}(O_k) and A_{vl} = A_1 \cup A_2 \cup A_3

where

A_1 = \{ < T, \text{tid(op)} > \mid \text{act(op)} = W, T \in V_{vl}(\text{vlbef(op)}),

T = \text{tid(op)}, \text{ and op} \in h_O \} \quad \text{--- CASE II-I}

A_2 = \{ < \text{tid(op)}, T > \mid \text{act(op)} = W, T \in V_{vl}(\text{vlaf}(op)),

T = \text{tid(op)}, \text{ and op} \in h_O \} \quad \text{--- CASE II-II}

A_3 = \{ < \text{tid(op)}, \text{tid(op)} > \mid \text{precedes(tid(op),tid(op))},

\text{rn(oid(op))} = \text{rn(oid(op))}, \text{ and }

\text{tid(op),tid(op)} \in V_l(O), \text{ op, op} \in h_l(O) \}

\text{--- CASE II-III. \square}

Algorithm 4 There is a directed graph, D_l(h) = (V_l(h), A_l(h)), for a complete history h.

Let D_l(O) = (V_l(O), A_l(O)) be a directed graph of an object, O, where O \in Obj(h). The set of vertices for D_l(h) is

V_l(h) = \cup V_l(O),

the set of arcs is

A_l(h) = \cup A_l(O). \square
Directed graphs created by algorithm 3 contain more edges than that created by algorithm 1. The additional edges, CASE I-III and CASE II-III represent the constraints imposed by local concurrency control.

**Theorem 3** There is a serial history $h_S$, of which a directed graph $D_S(h_S)$ is equivalent with $D_l(h)$ if $D_l(h)$ is acyclic.

**Proof.** As in theorem 1, there is a partial order $P_S$ in $D(h)$ by lemma 1 and let $h_S$ be the serial history which preserving $P_S$. Two histories $h$ and $h_S$ contain the same operations. Therefore all vertices in $D_l(h)$ exists in $D_S(h_S)$ and vice-versa.

Let us show there are the same arcs in both $D_l(h)$ and $D_S(h_S)$. Firstly, arcs created by CASE I-I, CASE I-II, CASE II-I, and CASE II-II are the same as in theorem 1. CASE I-III and CASE II-III: because $D_S(h_S)$ preserves the given partial order $P_S$, transactions precede other transactions in $D_S(h_S)$.

Second, let us assume that an arc$(u,v)$ exists in $D_S(h_S)$ but not in $D_l(h)$. Arcs created by CASE I-I, CASE I-II, CASE II-I, and CASE II-II are the same as in theorem 1. CASE I-III and CASE II-III: operations of two transactions $u$ and $v$ must be preserved in both $h$ and $h_S$ and the arc$(u,v)$ does not exist in $D_l(h)$, therefore transaction $v$ precedes transaction $u$ in $D_l(h)$ because only the sequence of operations can be changed in $h_S$. This is a contradiction because $h_S$ must preserve the given partial order. □

**Theorem 4** The history $h$ is serialisable if $D_l(h)$ is acyclic.

**Proof.** By theorem 3, there is a equivalent serial history $h_S$ for $h$ if $D(h)$ is acyclic, and by definition 6 and 7, a history $h$ is serialisable. □

5.3 The User-defined Concurrency Control Object

Until now, we have assumed an object is sharable unless its value is replaced to maximise concurrency in the system. But this form of sharing may be undesirable in some applications. The user-defined concurrency control object is introduced for such applications. Any partial changing of the state of such an object is treated as a replacement.
**Definition 9** An object is a user-defined concurrency control object if the object is a composite object and defined as an exclusive object by owners of the object. An element of the version list of the object consists of a set of pairs of an object identifier and an version number. These object identifier and version number indicate on which object actual replacements were happened due to an operation. □

**Definition 10** For an operation, op, on a user-defined object, O, its action is classified as read, R, if all terminal operations created by the operation, op, have no write action; otherwise is classified write, W. □

The algorithms described in section 5.2 can be used to create a directed graph of a complete history which includes user-defined concurrency control objects.
Chapter VI
Recovery

In this chapter, we describe a recovery scheme appropriate for our model. Recovery guarantees failure atomicity. Our recovery scheme is concerned with two aspects of failure atomicity: transaction failure atomicity and system failure atomicity. The possibility of transactions aborting requires the ability to restore consistent states of objects, and the possibility of system failure requires the ability to recover the effects of committed transactions after a system crash.

Transaction abortion occurs only in ASOSs and system failure occurs in both ASOSs and PSOSs. We provide both active and passive recovery. Active recovery means recovery from transaction abortion and system failure in ASOSs. It restores consistent states of objects in the case of transaction abortion and refetches objects from PSOSs in the case of system failure. Passive recovery means recovery from system failure in PSOSs. It reflects the effects of all transactions which had committed at the time of the crash. Passive recovery is designed to save file space and to make recovery fast.

6.1 Atomic Actions and System Recovery

Recovery schemes to ensure the failure atomicity of a system had been developed to ensure reliability of systems before the concept of transaction appeared. The well-known recovery schemes are logging [Gray 79] [Schwarz 84] (intention list [Lampson 81] or audit trail [Verhofstad 78]), shadows (backup/current version) [Lorie 77] [Gray 81], multiple copies [Verhofstad 78] and differential file [Severance 76] [Aghili 82]. In a logging scheme all update operations are logged out onto a log file or intention lists. Shadow schemes maintain two copies of the database and unlike
logging updates are done in place. Multiple copies uses several redundant copies of the database and all copies are of equal status. In the differential file scheme, the original copy of the database is kept in the base file and changes are stored into differential files.

To ensure the atomicity of transactions, we require recovery from transaction abortions as well as system crashes. The traditional recovery schemes can be used for this purpose. For this purpose modified data must be saved before transactions are completed. During the recovery, the effects of completed transactions must be reflected into the recovered states of data and the effects of uncompleted transaction must be removed. We divide recovery tasks into two parts: active recovery which restores consistent states of objects due to transaction abortion and refetches replicas of objects from a PSOS in case of system failure and passive recovery which reflects the effects of committed transactions into the recovered objects when a system failure occurs.

We propose a passive recovery scheme which cooperates with our cache coherence protocol. The proposed passive recovery scheme is based on a differential file and a shadow copy. The differential file is used mainly for recovery of PSOSs and supplying version information to replicas in ASOSs. The differential file consists of a base version and history entries of services. History entries have dual use: for version information and for recovery information. The shadow copy is the merged version of the base version and history entries; it has the latest version of groups of objects in the PSOS. The shadow copy is used to reduce the amount of data required to load a service from PSOSs to ASOSs and to improve the access time by avoiding access of the differential file. Note that the shadow copy is not the same as the shadows proposed by [Lorie 77] [Gray 81].

6.2 Active Recovery

In ASOSs, transaction abortion and system failure can happen. When a transaction is aborted, affected objects restore their consistent state from their version information in ASOSs. Each object replica in ASOSs has its version information. When a transaction is committed, affected objects save their new version in PSOSs and other replicas of corresponding objects in other ASOSs are sent the same information. When a transaction
is aborted, affected replicas restore their states according to their latest version.

If an ASOS has crashed, all transactions and active services in the ASOS fail. However, such a failure only affects the users of the affected ASOS unless transactions in other nodes use a service in the failed ASOS. If such transactions exist, they have to restart with another replica of the service in another ASOS. If there is no replica of the service in another ASOS, their restart will be delayed until the ASOS is recovered. Active replicas of services in the failed ASOS are reloaded from PSOSs (see section 6.3).

When an ASOS starts its recovery action, it tries to recover its state before the failure happened, so that users can restart their tasks without requesting all objects again. In other words, we prefer warm caching to cold caching after the recovery. The recovery action of the failed ASOS starts with broadcasting its failure and a request to PSOSs to find out which services were used by the ASOS before it failed. PSOSs send the latest version of replicas of services which were in the ASOS when the last contact was made with the ASOS before it crashed. Active services in ASOSs are temporary, so losing them does not cause any consistency problem - the consistency of services is maintained in the context of transactions (see chapter 4 and 5). Apart from the few transactions which are using any service in a ASOS, the loss of the ASOS does not affect other ASOSs (see section 4.2).

6.3 Passive Recovery

Passive recovery deals with recovery from service failure. Losing a PSOS would mean losing all services in the PSOS permanently. Therefore, a PSOS must have the ability to restore its state after a crash. At least one PSOS provides a differential file which consists of a base version of a group of objects in a service and history entries which contain the history of all changes made to the group. The base version and history entries are saved on stable storage. Stable storage guarantees that any objects stored are not lost unless disaster [Lampson 81] happens.

In addition, the PSOS provides a shadow copy of the group of objects. The shadow copy is the latest version of the group and is the same as the
merged image of the base version and history entries. When a replica in an ASOS loads a service, it uses the shadow version to avoid merging the base version and history entries to calculate the latest states of objects. In addition, the shadow copy serves as a replicated service. When stable storage is damaged by a disaster, the shadow copy can be used as the back-up copy of the service if it is stored at a different storage location because it always has the latest version of the service before the disaster happened.

Note that no transaction can be committed after the disaster has occurred because it cannot get a reply from the PSOS which stores the differential file.

The state of all the objects in the distributed system is logically replicated in several PSOS replicas. Some of these replicas have both a differential file and a shadow copy. Others have just a shadow copy. Only PSOSs having the differential file are recoverable from system crash, and other PSOS restores their states from them. In this chapter we describe recovery of a PSOS from the differential file. However if the PSOS is replicated, recovery would also involve requesting recent updates from another PSOS replica. For example, if Quorum Consensus replication is in use, outdated replicas will automatically acquire new versions of objects from other PSOS replicas that are more up-to-date.

Note that a PSOS is not partitioned in our model and if a PSOS is replicated, the entire PSOS is replicated.

A service which is replicated into several PSOSs and ASOSs. The replicas form a cluster. Each member of a cluster knows about the other members and can communicate with them by multicast.

A PSOS has for each service which it stores a list of ASOSs which has any replica. It also has a list of all transactions which are in commitment phase or have recently committed or aborted containing the status of each transaction (either prepared, committed or aborted). For each prepared transaction it has the prepared updates to objects and an intentions list referring to these updates. Both lists are stored on stable storage.
When a transaction starts commitment, replicas affected by the transaction in ASOSs send a 'prepare-commit' message to PSOS replicas which store their passive objects. The message includes all changes made by the transaction and its identifier. After receiving the message, each PSOS, involved in the commitment, adds the transaction identifier into its transaction list, marks it as 'prepared' and stores the changes on its stable storage.

If the transaction can commit, the backup coordinator sends a 'commit' message to all the PSOS replicas. Each PSOS replica marks the transaction as 'committed' and confirms the commitment. Then it appends the changes made by the transaction into its differential file. This creates a new history entry in the differential file. At the same time, the shadow copy of the differential file also updates its states according to the changes made by the transaction. During the updating of the shadow copy, its access is blocked, but the confirmation of the commitment of the transaction is not affected because any failure during shadow update can be restored from the differential file. Note that the shadow copy updates its state only after it sees commitment confirmation.

When a PSOS recovers it examines its transaction list and checks whether all changed made by 'committed' transactions in the list is completely appended in the differential file.

If not, the PSOS deletes the incomplete history entries and re-approves the changes until it succeeds. After appending the PSOS multicasts a 'committed' message (see chapter 4). Until the PSOS recovers, transaction commitment is generally blocked, but this depends on the replication method used by PSOS replicas. If Quorum Consensus is in use, then the commitment will require a write quorum on the difference file and on the shadow copy.

If the system failure occurs during the shadow copy updating, the shadow copy must be recreated from the differential file by merging the base version and history entries. During the time, ASOSs have to access the differential file to load an service they want. In this case, ASOSs have to merge the base version and history entries of the service.
The base version and history entries are merged periodically. This results in the creation of a new base version. After creation of the new base version, it replaces the old base version and history entries. When a system crash has happened during the merge, the PSOS discards the incomplete version and re-merges the old version and history entries. The merge procedure is described at section 6.4.

History entries are also used to provide version information for replicas in ASOSs; enabling them to supply the missing version of objects when they are required by replicas in ASOSs. Because this information is used by the passive recovery scheme, storage space is saved by this dual usage of the history entries (see chapter 4).

6.4 The Recovery Scheme

The state of a service is moved from one consistent state to another consistent state by a transaction $T_i$, i.e. $S_i = T_i(S_{i-1})$ where $T_i$ transforms state $S_{i-1}$ into state $S_i$. The transformation can be seen as adding, deleting, and updating objects in the service, i.e. $S_i = (S_{i-1} + A_i + U_i) - D_i$ where $A_i$ contains objects added by transaction $T_i$, $U_i$ contains objects modified by $T_i$, and $D_i$ contains objects deleted by $T_i$. Let us use $H_i$ for the modification done by $T_i$, i.e., let $A_i + U_i - D_i$ be $H_i$. Then $S_i = S_{i-1} + H_i$. To transfer the consistent state $S_{i-1}$ to $S_i$, $H_i$ is only required. If an ASOS already has $S_{i-1}$, then it can use the knowledge of $H_i$ supplied by the PSOS to create $S_i$. This avoids reloading the current state of the service from the PSOS into the ASOS. Moreover $H_i$ is created by an ASOS in which the transaction $T_i$ was executed, and the internal structure of $H_i$ is made suitable for transmission via a computer network and is saved onto PSOSs without further processing. The version number of a service is increased by one whenever a transaction commits. The structure of a service in a PSOS is shown in figure 6.1(a).

Two copies of a service are stored in a PSOS replica. One is a form of differential file and the other is a shadow copy. The differential file consists of a base version of a service and history entries, containing modifications. The base version and its history entries together are called the original copy. The symbol, O(S), is used to represent the original copy
of a service $S$ and the symbol $S(S)$ stands for the shadow copy of the service $S$. The subscript, $n$, represents the version number of the copy.

$$O_n(S) = B_0 + H_1 + H_2 + ... + H_n$$
$$S_n(S) = O_n(S)$$

where $n$ is the version number of the service $S$.

When a modification $H_{n+1}$ arrives at the PSOS, $H_{n+1}$ is appended to its original copy and then $S_n(S)$ is modified to create $S_{n+1}(S)$. As soon as the appending is done, the original copy sends a commitment confirmation to the corresponding objects which sent the $H_{n+1}$. The confirmation message triggers the modification of the shadow copy and replicas in ASOSs. When a system crash occurs during appending, the recovery process removes $H_{n+1}$ if it is partially appended; otherwise it sends the confirmation message to update the shadow copy and replicas in ASOSs. Even if the original copy is inaccessible due to a crash, the service is partially available through the shadow copy for some read-only transactions and read-write transactions, but read-write transactions cannot commit. A read-only transaction using the same version as that of the shadow copy can progress because it can obtain all necessary information via the shadow copy. If the service is replicated then the recovery process takes missing parts of history entries from other replicas in other PSOSs. In this case the confirmation message is not required and no active transactions are blocked in their commitment.

The shadow copy is introduced to avoid the disadvantage of differential files. When a differential file is used, accessing data from the differential file is a complex operation: both the original copy and differential file must be accessed to find out the correct state of data. The shadow copy simplifies this operation because the state of the shadow copy is the latest version of the objects: it improves the speed of accessing the state of the service in PSOSs. Another advantage is that merging the base version with history entries is not required when the size of history entries becomes large. Instead of merging them, we copy the shadow and make the copied version a new base version of the service. During copying the shadow in order to create a new base version or to load parts of the service, updating the shadow is not permitted. Due to this blocking the shadow
becomes outdated. But this problem can be solved by supplying the update information, which is not consumed, together with the out-dated shadow; the out-dated replica is updated at ASOSs. Note that creating a new base version does not block transaction commitment. The two operations are independent and can be executed concurrently.

After copying the shadow, the original copy is garbage collected. But some of the history entries must be kept because the may be required by replicas in ASOSs. The PSOS searches backwards in the history entries until it has found a history entry created by each of the replicas. Earlier history entries may be garbage collected. Each history entry has a list of version numbers, \( H_v \) and an identifier of a replica, \( R_k \). The identifier \( R_k \) is assigned by the PSOS manager and kept in the replica list of the service, \( S_R \). \( S_R \) contains all identifiers of replicas currently in use at ASOSs. Let us suppose that there are \( n \) history entries for an group of objects in a service, \( H_1 \rightarrow H_n \). A set of identifiers of replicas, \( R(H_i) \), is defined as the set of identifiers of replicas used by transactions which generates history entries from \( H_i \) to \( H_n \):

\[
R(H_i) = \{ R_k \mid H_j.R_k \text{ and } 1 \leq i \leq j \leq n \}.
\]

Then there will be a history entry \( H_i \) of which \( R(H_i) \) is the same as \( S_R \), i.e., \( R(H_i) = S_R \). This means that all replicas in ASOSs have more up-to-date versions than that of \( H_i \). All replicas in \( R(H_i) \) were committed at least once after \( H_{i-1} \); otherwise they cannot included in \( R(H_i) \) by definition.

When each replica committed, it had the latest version of the group at that time. Therefore, all replicas in \( R(H_i) \) have more up-to-date versions than that of \( H_i \). Because \( R(H_i) \) is the same as \( S_R \), all replicas in ASOSs have more up-to-date versions than that of \( H_i \). Therefore, history entries from \( H_1 \) to \( H_{i-1} \) are garbage collectable without consulting replicas in ASOSs. Read-only transactions also notify PSOSs of their version numbers when they are committed - they are always committed.
An example of the base version and history entries of a group is shown in figure 6.1(a). There are three replicas in ASOS, i.e., $S_R = \{ R_1, R_2, R_3 \}$, and four history entries, $H_1 - H_4$. $R_S(H_4)$ is $\{ R_1 \}$, $R_S(H_3)$ is $\{ R_1, R_2 \}$ and $R_S(H_2)$ is $\{ R_1, R_2, R_3 \}$ and $R_S(H_1)$ is the same as $R(H_2)$. When we create a new base version, $H_1$ can be garbage collected because the condition, $S_R = R(H_1)$, is met by $R(H_2)$.

Note that garbage collection can be executed only when a new base version is created; otherwise recovery from system failure cannot be guaranteed. The total size of the history entries depends on the speed of the slowest transaction using one of replicas because the change made by a transaction is always reported to a PSOS whenever the transaction is completed. If a transaction is very slow in using a replica - or an ASOS having a replica of the service has crashed and takes very long time to recover - and other transactions using different replicas are very fast, then the total size of the history entries can be very large because there is no $R(H_1)$ satisfying the condition $S_R = R(H_1)$. Therefore, such replicas may be a major obstacle to garbage collection.
The PSOS that possibly has a service involved in a long transaction may ask each of the replicas in ASOSs what is the latest version number they have. Then the PSOS can force replicas to be updated if they are out-dated. In the case that an ASOS has crashed, the PSOS can remove the identifier of the replica in the ASOS from $S_R$ until the failed ASOS recovers. These replicas can be calculated by $S_R - R(H_i)$. For example, Let $S_R = \{ R_1, R_2, R_3 \}$ and a transaction using $R_1$ is very slow, then history entries will grow as in figure 6.1(b).

6.5 An Example of Recovery

In this section an example of our recovery scheme is shown. Let us assume that a service consists of two composite objects A and B and a simple object C. Each composite object has two simple objects, D and E and F and G respectively. The service has two replicas in ASOS$_1$ and ASOS$_2$. There are four transactions, $T_1$ - $T_4$. $T_1$ and $T_2$ use the replica in ASOS$_1$ and $T_3$ and $T_4$ uses the replica in ASOS$_2$. All replica objects are loaded from the base version.

Figure 6.2 shows the initial structure of the service in ASOSs and a PSOS. Object C, D, E, F, and G belong to the integer class and object A and B belong to the point class. The shadow copy of the service is omitted in the figure. Initial states of objects are $A( D[1], E[0] )$, $B( F[10], G[10] )$, and $C[20]$. Subscripts of objects represent the replica number. The subscript represents corresponding replica number used by a transaction and the number in square brackets represents the value in the replica object.
The Base Version (version 1)

<table>
<thead>
<tr>
<th>Object A</th>
<th>Object D</th>
<th>Object E</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS POINT</td>
<td>CLASS INTEGER</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object B</th>
<th>Object F</th>
<th>Object G</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS POINT</td>
<td>CLASS INTEGER</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object C</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS INTEGER</td>
</tr>
</tbody>
</table>

History Entry [H1:R1] (created by the replica in ASOS1 for the transaction T1)

<table>
<thead>
<tr>
<th>Object E</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS INTEGER</td>
</tr>
</tbody>
</table>

| UPDATED | ADDED | DELETED |

History Entry [H2:R1] (created by the replica in ASOS1 for the transaction T2)

<table>
<thead>
<tr>
<th>Object F</th>
<th>Object C</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS INTEGER</td>
<td>CLASS INTEGER</td>
</tr>
</tbody>
</table>

| UPDATED | ADDED | DELETED |

History Entry [H3:R2] (created by the replica in ASOS2 for the transaction T3)

<table>
<thead>
<tr>
<th>Object F</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS INTEGER</td>
</tr>
</tbody>
</table>

| UPDATED | ADDED | DELETED |

Figure 6.2 An example of a base version and history entries
The sequence of events is:

1: \( T_1 \) reads \( D_1 \) (0)
2: \( T_2 \) writes \( F_1 \) (40)
3: \( T_3 \) writes \( F_2 \) (50)
4: \( T_1 \) writes \( E_1 \) (30)
5: \( T_1 \) committed
6: \( T_4 \) reads \( C_2 \) (20)
7: \( T_2 \) deletes \( C_1 \)
8: \( T_2 \) committed
9: \( T_3 \) reads \( E_2 \) (30)
10: \( T_3 \) committed
11: \( T_4 \) committed

Until event 4, all three transactions executed without any conflict. Because \( T_2 \) and \( T_3 \) use different replicas, they don’t conflict due to event 3. After \( T_1 \) committed [event 5], the replica in the ASOS_2 changes its state according to the information sent by the replica in ASOS_1.

Note that the information is sent as part of the two-phase commitment protocol. In this case the history entry \( H_1 \) containing the object \( E \) is modified, i.e., integer 10 is replaced by another integer 30. Now, both replicas of \( E \) have version number 2 - we assume the base version has version number one.

Events 6 and 7 are done without causing a conflict because these operations are done on the different replicas. After event 8, object \( F \) is replaced (integer 20 -> integer 40) and object \( C \) is deleted. This information is received by the replicas of objects \( C \) and \( F \) in ASOS_2. It causes the abortion of \( T_4 \) because \( T_4 \) read object \( C \); \( T_4 \) caused a read-write conflict. \( T_3 \) is not affected because the writing operation (event 3) is considered overwriting object \( F \) created by \( T_2 \) rather than overwriting the base version. In other words, the current version of the replica in ASOS_2 becomes \( H_3 \) and the replica of object \( F \) has the temporary value 50 until \( T_3 \) commits. The value of object \( F \) becomes 10 for the base version and \( H_1 \), 40 for \( H_2 \) and 50 after \( T_3 \) committed. Note that the version number of object \( F \) is increased when \( T_1 \) is committed because all objects in the same group have the same version number.
At Event 9, T3 reads object E[30] created by T1 if the version number of the replica in ASOS2 is greater than one - it already updated its version according to the information sent by the replica in ASOS1 even if it does not receive the information for the version three created by T2. After event 10, if a new base version is created, then the first history entry is garbage collectable because the PSOS knows both replicas have at least version two, i.e., \( R(H_2) = S_R = \{ R_1, R_2 \} \).

Let us assume that ASOS1 failed after the commitment of T1. This does not affect T3 and T4 because they use the replica in ASOS2 and use no service replicas in ASOS1. Firstly, let us assume that the replica in ASOS2 did not receive the 'prepare-commit' multicast message from the replica in ASOS1 because it was lost. When T3 tries to commit, the PSOS recognises that the replica used by T3 has the out-dated version. ASOS2 updates its version from the history entry, which is stored in the PSOS and was created by the replica in ASOS1 before the crash and after the commitment of T1. T3 is aborted because it read out-dated state of object E - a read-write conflict. T3 can be restarted immediately because the replica in ASOS2 has the latest version now. T4 can commit now because T2 is aborted due to the failure in ASOS1.

Secondly, let us suppose that the replica in ASOS2 receives the commitment message of T1. Now T3 can commit now because the read-write conflict in the previous case does not occurs; because object E updated its state before T2 read, the read-write conflict is prevented. Because T2 in ASOS1 is aborted due to ASOS1's failure, T4 can commit as well.

We now consider the failure of the PSOS during the transaction commitment. When the PSOS fails during the commitment of T2, all commitment of transactions using the service is blocked until the service is restored. During the recovery of the PSOS, the list of transactions of the PSOS is checked. If T2 is marked as 'committed', the history entry for T2 is examined. If the appending of the modification information sent by T2 is not completed, the appending procedure is repeated after removing the uncompleted history entry. If T2 is marked 'uncommitted', the PSOS does nothing unless it is the back-up coordinator of T2 (see chapter 4). The
entire reconstruction of shadow copy is required if the failure occurred during the modification of the shadow copy but this can be avoided if the updating process uses careful history entries [Verhofstad 78] - keeping another copy during updating and removing the old copy after updating. However, commitments of transactions are resumed as soon as the PSOS can handle appending operation to its differential file.
Chapter VII

Deadlock Detection

Concurrent processes can share resources, but when a non-sharable resource is used by a process and another process may want to access it, the latter must be blocked to prevent interference between them. The blocking is resolved when the holding process releases the resource. Cyclic waiting may occur due to the holder of a resource, blocking accesses of other processes, and waiting on another resource whose access is blocked by another process. As a result, the processing of all processes in the cycle cannot ever be done. Such a situation is called deadlock. For example, when locking is used to control consistency among concurrent transactions, a request to access an object by a transaction may be granted if no other transaction holds a lock on the object; otherwise the transaction is blocked. If two transactions, T1 and T2, hold two objects, O1 and O2, respectively, and want to access the other objects respectively, then the two transactions are in a deadlock, because no transaction can acquire both objects—neither transaction can proceed.

Detecting deadlocks in structured entities, objects, requires more careful consideration than non-structured ones:

- The structure of an object can be seen as a graph structure; an object has other objects as its states in an object-oriented system. Therefore, finding a deadlock may require traversing all the graph structure. For example, let us assume that an object A is replaced by another object B, and the object A contains two instance variables A1, and A2. The conflicts between transactions happen not only in A but also in A1 and A2. In other words, to detect the wait-for relation between transactions is not as simple as in non-structured ones.
Most objects are not used by transactions directly. In the above example, A1 and A2 are accessible only via A, therefore, a transaction using A has no information as to whether it uses A1 or A2.

There can be a lot of objects in the system. A lot of memory may be consumed if all the objects try to keep deadlock detection information - a deadlock detection algorithm should minimise the amount of memory it requires. Therefore, avoiding duplicate information in objects is very important.

Because services are replicated in our model and transactions using different replicas cannot cause deadlock, the probability of deadlock is reduced in proportion to the number of replicas, if there is an equal probability of a transaction using any replica of a service, but the introduction of replicas increases likelihood of transaction aborts. Distributed deadlock detection detects deadlocks between ASOSs in our model. Because ASOSs are usually located in different nodes, reducing communication traffic is also very important, since communication cost is the major cost in detecting distributed deadlocks. In this chapter we use reliable unicast (point-to-point) communication unless we mention explicitly otherwise.

In our model, deadlock detection is divided into two parts: local and distributed deadlock detection. PSOSs are not involved in either of these tasks.

Our local deadlock detection detects deadlocks within ASOSs - Each ASOS has its own local deadlock detector. It maintains a local transaction wait-for graph and detects local deadlocks. The graph is kept as a set of edges which represent direct and indirect dependencies between transactions. Both the local and the distributed deadlock detector use an edge-chasing method. The difference between local and distributed detection is that local detection traces edges in both directions, forward and backward, because it has no communication overhead, whereas the distributed detection traces edges in one direction, forwards, to reduce communication cost. By forwards we mean in a direction from a
transaction that is waiting to access an object to another transaction that holds the object.

7.1 Relationship to Prior Work

A deadlock involves transactions and the objects which are held by them. Transactions are blocked by objects due to the consideration of the consistency of their states. Therefore a deadlock can be detected by communication between transactions and objects - such as Sihna and Natarajan’s [85-1] and Roesler and Burkhard’s [88] algorithms.

In both of these algorithms, a transaction sends its request messages to its transaction manager (TM) and the transaction manager forwards the message to an object’s manager (OM), which manages an object which the transaction wants to access, and then the object manager forwards the message to the object – see Figure 7.1(a). Deadlock detection messages are generated by OMs when OMs detect antagonistic edges (see section 7.3) in their wait-for graph. These messages are forwarded from OMs to TMs and from TMs to OMs until deadlocks are found or deadlock detection messages cannot be forwarded because they are arrived at a TM/OM which has no outgoing edge. When a deadlock is found, the deadlock is resolved by aborting a transaction involved in the deadlock and removing all edges related with the aborted transaction by sending clean-up messages via the same route taken by the deadlock detection messages. The clean-up messages are also created when a transaction is completed - it is removed from a transaction wait-for graph (see section 7.2-7.3).

Both of these algorithms have three problems:

- **Transactions may forward deadlock detection messages unnecessarily.** When a transaction requests an object in a service, the transaction has no idea whether the request is granted but the processing is so slow that it has not received any message yet; or it is blocked on the object. If the granting processing is slow, the messages are regarded as *meaningless* and discarded.

- **All messages must pass though transaction managers.** To send a deadlock detection message from an object manager to another
object manager, two messages are required: the first from one object manager to a transaction manager and the second from the transaction manager to another object manager. The reason is that only a transaction manager of a transaction knows all out-going edges which include the transaction.

A transaction does not know exactly in which object it is blocked in an object-oriented environment. As a result, a deadlock detection message has to travel from an object directly accessed by the transaction to the object in which the transaction is blocked. In other words, there is not only communication between transaction managers and object-managers but also communication between object managers.

![Diagram](image)

(a) A transaction manager
(b) A local deadlock detector

Figure 7.1 (a) A message follow (a) in Sihna and Natarajan's [85-1] and Roesler and Burkhard[88]'s algorithms, and (b) in our algorithm

Our deadlock detection algorithm\(^1\) is an extended version of the Sihna and Natarajan's [85-1] and Roesler and Burkhard[88]'s algorithms for use in an object-oriented environment as well as to lessen the above problems. We introduce coordinators instead of the object managers and transaction managers suggested in both algorithms. Each transaction chooses one of the local detectors as its coordinator. The coordinator

---

\(^1\)The deadlock detection algorithm is a updated version of the author's algorithm suggested in [Min 89] to apply to distributed object-oriented systems.
receives all deadlock detection messages on behalf of transactions. When a transaction is blocked on an object, this is reported to its coordinator rather than the transaction manager. The report indicating a transaction dependency triggers a local deadlock detection. If the coordinator, which is a local deadlock detector itself, finds a cycle due to the new edge, transactions involved in the cycle make a deadlock.

The local deadlock detector for each ASOS has a partial wait-for graph, which is in form of edges (probes), of the entire wait-for graph which is the combined wait-for graph of all local deadlock detectors. A transaction may use services in more than one ASOS. Therefore, transactions may cause deadlock across ASOSs. Such deadlocks are found via communication between local deadlock detectors. If the transaction is blocked on an object, the object sends an edge representing the blocking to its local deadlock detector. If the local deadlock detector is not the coordinator of the transaction, it forwards the edge to its coordinator.

When the coordinator of the transaction inserts the edge in its wait-for graph, it can find out new dependency relationships which include the new edge. Note that the new edge causes indirect dependencies as well as direct dependencies. The coordinator initiates a distributed deadlock detection message if there is a new dependency, of which the coordinator of the destination transaction is not the coordinator, and the dependency causes an forward (or backward) waiting between the origin and the destination transaction.

Most of the algorithms use transaction priorities which gives an ordering on transaction identifiers. Deadlock detection messages are sent only from transactions with higher priority to those with lower priority. This priority scheme reduces the number of messages required for deadlock detection by half; lexical ordering is used by Obermark [82] and priority is used by Sihna and Natarajan [85-1] and Roesler and Burkhard [88].

The concept of coordinators in our algorithm replaces transaction managers. It removes the necessity of communication between transaction managers and object managers; it reduces the number of messages to be sent to detect distributed deadlocks. It also reduces the amount of storage to store the deadlock detection information because it avoids the need to
store the same edges at TMs and OMs. Furthermore, objects report dependencies, but no objects are involved at deadlock detection. Local deadlock detectors detect deadlocks based on the dependencies reported by objects, so the meaning of dependency is not related to deadlock detection.

How to detect dependencies and which dependencies are considered as a blocking dependency are up to objects (or their policies). For instance, dependency can be based on semantic information on objects, and such information is object dependent. This does not affect our detection algorithm. In other words, our deadlock detection algorithm is independent of concurrency control. Let us compare two concurrency control policies. If an object uses two-phase locking, replacing it with another object which is in use by another transaction immediately causes a conflict dependency, i.e., a blocking. Therefore, the object reports the conflict. However if the object uses multiversion locking, such an operation does not cause a conflict dependency immediately because the new object is considered as a new version. In this case, the object does not report the conflict even if the conflict may indeed occur when the transaction tries to commit.

The algorithm we propose has the same properties as those of Obermark's algorithm [82] as well as Sihna and Natarajan[85-1] and Roesler and Burkhard[88]. The latter two algorithms are designed for object-based environments:

- We use local and distributed deadlock detectors as in Obermark's algorithm [82].
- Deadlock detection messages are optimised.
- The deadlock detection algorithm is independent from concurrency control as in Obermark's algorithm [82].
- It is easy to calculate the cost of deadlock detection.

Our algorithm improves the algorithms of Sihna and Natarajan[85-1] and Roesler and Burkhard[88] in three ways:
1. It uses a simpler model and reduces the number of deadlock detection messages by up to half those generated by both algorithms,
2. It uses less memory, and
3. It is designed to deal with structured objects.

7.2 The Structure of a Local Deadlock Detector

Transactions communicate with objects via a message passing mechanism. Objects can be accessed indirectly by transactions via other objects. If there is a conflict between two transactions on objects, the object reports the conflict to its local deadlock detector. Each transaction has its own transaction identifier which is unique across ASOSs.

As in Figure 7.2, each local deadlock detector has one coordinator table and one transaction wait-for graph (TWFG).

Transaction Coordinator’s Table stores coordinators of transactions which use at least one service in the ASOS. A new element is created when a transaction starts to use a service in the ASOS. The coordinator of the transaction is registered when a transaction is created in the ASOS or when a transaction from another ASOS uses its service in the ASOS for the first time and it is removed when it is committed or aborted.

Transaction Wait-for Graph contains edges supplied by local objects in which a transaction, whose coordinator is this local deadlock detector, is blocked by another transaction or supplied by other local deadlock detectors. Edges represent indirect as well as direct dependencies between transactions. An edge consists of a pair, each of which specifies a transaction identifier and its coordinator’s address. The first transaction is waiting for the second transaction. It also contains a counter field to show how many paths exist between two transactions.
7.3 Probes: A Priority Scheme

In our algorithm, deadlocks are detected by detecting cycles in the transaction wait-for graph. An edge of the transaction-wait-for graph consists of a pair of a transaction identifier and its coordinator and a counter, i.e. $<T_1:C_1, T_2:C_2, \text{Counter}>$. $T_1$ is blocked by $T_2$. The counter represents the number of different paths between two transactions in the wait-for graph. If a transaction identifier $T_i$ is greater than another transaction identifier $T_j$, i.e. $T_i > T_j$, $T_i$ has higher priority than $T_j$. An edge $(T_i, T_j)$ is called an antagonistic edge if $T_i > T_j$. A deadlock detection message is created when an antagonistic edge is inserted into the transaction wait-for graph of a coordinator. We call a deadlock detection message sent by a local deadlock detector a probe. Each probe message inserts an antagonistic edge in the receiving coordinators' graph. The priority scheme was first suggested by Obermark[82] and also used by Sihna and Natarajan[85-1] and Roesler and Burkhard[88].

An edge represents a direct or indirect wait-for relationship between two transactions. An indirect wait-for relation may cease due to resolution of a direct wait-for relationship. For example, let us assume three transactions...
have the following wait-for relationships, $T_1 \rightarrow T_2 \rightarrow T_3$, when the direct wait-for relationship, $T_2 \rightarrow T_3$, is resolved then the indirect wait-for relationship, $T_1 \rightarrow T_3$, exists no longer. Therefore when an indirect wait-for relationship is resolved, edges representing non-existing relationships must be removed from the wait-for graphs of coordinators involved. A message created to remove such indirect relationships is called a clean-up messages or an anti-probe.

### 7.4 Deadlock Detection and Resolution

When an object finds a dependency between two transactions, $T_i$ and $T_j$, it sends an insert_edge($T_i$:L, $T_j$:L, 1) message to its local deadlock detector (LDD$_i$), and when the dependency is resolved, a remove_edge($T_i$:L,$T_j$:L,1) message is sent to its local deadlock detector (LDD$_i$). After receiving the insert_edge (remove_edge) message, the local deadlock detector inserts (or removes) an edge if it is $T_i$'s coordinator; otherwise it forwards the message to $T_i$'s coordinator. The coordinators of all transactions using any service in the ASOS are stored at the transaction coordinator table (TCT) of the local deadlock detector.

The local deadlock detector which receives the insert_edge message finds out new dependencies using the local_dependency procedure (see Figure 7.3 and 7.4). The procedure uses the edge-chasing method to find out local dependencies. These new dependencies and the edge are inserted into its transaction wait-for graph (TWFG), and then it checks whether the inserted edge causes a local deadlock cycle. If it finds a deadlock cycle it aborts the transaction having the highest priority in the deadlock cycle; otherwise it finds antagonistic dependencies (edges) created by the new edge. If the coordinators of the destinations of these antagonistic edges are not local then it forwards the edge in a probe message to their coordinators because these edges may cause deadlock cycles.

When a remove_edge message is received, the local deadlock detector removes dependencies that no longer exist. The local deadlock detector traces local dependencies using the local_dependency procedure. The traced local dependencies are removed from its TWFG. It also forwards a remove_edge message for each of the antagonistic dependencies found in local dependencies. Because such edges are inserted by the insert_edge
procedure, they must be removed to prevent detection of phantom (non-existing) deadlocks. These messages are called anti-probe messages.

The local_dependency procedure (Figure 7.4) traces all new dependencies created by the insertion of a new edge. It finds out all adjacent edges with edges in new_dependencies[1]. The number of iterations of the while loop is the same as the actual length of the edges in new_dependencies[1] between two transactions in TWFG. For example, when we assume the inserted edge is \(<T_1, T_2>\) and the TWFG contains \(<T_5, T_1>, <T_2, T_3>, \) and \(<T_2, T_4>\). At the first iteration, there is only one edge, \(<T_1, T_2>\), in new_dependencies[1], whose length is one. At the second iteration, there are three edges, \(<T_5, T_2>, <T_1, T_3>, \) and \(<T_1, T_4>\), in new_dependencies[1], whose length is two. At the third and final iteration, there are two edges, \(<T_5, T_3>\) and \(<T_5, T_4>\), in new_dependencies[1]. Note that all six edges are inserted into TWFG.
LDD:insert_edge( <T_i;C_i, T_j;C_j, 1> )
new_local_dependencies : EdgeStack;
begin
  if C_i = ⊥ and C_j = ⊥ then
    /* an insert request sent by a local object */
    C_i := TCT(T_i); C_j := TCT(T_j);
    if TCT(T_i) ≠ LDD_i then
      /* forward an insert_edge message to the T_i's coordinator */
      send insert_edge(<T_i;C_i, T_j;C_j, 1>) to TCT(T_i);
      return
    endif
  else if <T_i;C_i, T_j;C_j, counter> in TWFG then
    increase counter in <T_i;C_i, T_j;C_j, counter>;
    return
  endif
  call local_dependency (new_local_dependencies, <T_i;C_i, T_j;C_j, >);
  if found_deadlock(new_local_dependencies) then
    call resolve_deadlock(new_local_dependencies);
  else
    join TWFG with new_local_dependencies;
    call insert_distributed_dependency (new_local_dependencies);
  endif
end return
end Insert_edge;

LDD:remove_edge( <T_i;C_i, T_j;C_j, 1> )
removed_dependency: EdgeStack;
begin
  if C_i = ⊥ and C_j = ⊥ then
    /* a remove request sent by a local object */
    C_i := TCT(T_i); C_j := TCT(T_j);
    if TCT(T_i) ≠ LDD_i then
      /* forward a remove_edge message to the T_i's coordinator */
      send remove_edge(<T_i;C_i, T_j;C_j, 1>) to TCT(T_i);
      return
    endif
  else if <T_i;C_i, T_j;C_j, counter> in TWFG then
    if counter > 1 then
      decrease counter in <T_i;C_i, T_j;C_j, counter>;
      return
    endif
  endif
  call local_dependency(removed_dependencies, <T_i;C_i, T_j;C_j, >);
  remove removed_dependencies from TWFG;
  call remove_distributed_dependency(removed_dependencies);
end remove_edge;

Figure 7.3  Deadlock detection algorithm - main procedures
Procedure local_dependency
(var new_local_dependencies:EdgeStack, NewEdge:Edge);
new_dependencies[1..2] : EdgeStack;
Inserted : Boolean;
Begin
  clear new_dependencies[1], new_dependencies[2];
clear new_local_dependencies;

  new_dependencies_found := True;
push NewEdge into new_dependencies[1];
while new_dependencies_found do
  new_dependencies_found := False;
  repeat
    pop Ed from new_dependencies[1];
    for all edges Eg in TWFG do
      /* forward dependency */
      if (Es.T2 = Eg.T1) and
      not (<Es.T1.C1, Eg.T2.C2, count>
in new_local_dependencies)
      then
        push <Es.T1.C1, Eg.T2.C2, 1>
        into new_dependencies[2];
        new_dependencies_found := True;
      /* Backward dependency */
      else if Es.T1 = Eg.T2 and
      not (<Eg.T1.C1, Es.T2.C2, count>
in new_local_dependencies)
      then
        push <Eg.T1.C1, Es.T2.C2, 1>
        into new_dependencies[2];
        new_dependencies_found := True;
      endif
    endfor
    push Ed into new_local_dependencies;
    until isEmpty(new_dependencies[1]);
  endwhile
  return
end local_dependency;

Procedure found_local_deadlock(new_local_dependencies:EdgeStack): Boolean;
begin
  for all edges Ed in new_local_dependencies do
    if Ed.T1 = Ed.T2 then
      return True;
  endif
endfor
end found_local_deadlock;

Figure 7.4 Deadlock detection algorithm - subsidiary procedures
The found\_local\_deadlock procedure (Figure 7.4) tests whether the insertion of a new edge caused a cycle - a deadlock. All new dependencies created by the insertion are stored in the new\_local\_dependencies. By comparing the origin and the destination of an edge, we can find whether there is a cycle or not. If the origin and the destination are the same then a cycle is found. This does not mean only one transaction (the origin and destination) is involved in a deadlock, but more than one transaction including the transaction may be involved in a deadlock. For example, let us assume that TWFG contains $<T_5, T_1>$ and $<T_2, T_5>$ and an edge $<T_1, T_2>$ is inserted. Then new dependencies, $<T_5, T_2>$, $<T_1, T_5>$ and $<T_5, T_5>$, are traced. Because $<T_5, T_5>$ is in the new dependencies, there is a transaction in the cycle, and three transactions are involved in the deadlock.

The following two procedures insert (remove) antagonistic edges between two local deadlock detectors. They trace edges in only one direction to reduce the number of messages to detect distributed deadlocks. These insert\_edge messages are treated in the same way as a local insert\_edge message; the receiving local deadlock detector traces its local dependencies created by the insertion and tests whether there is a cycle, and then it forwards antagonistic edges to another local deadlock detection until no probe (antagonistic message) is created or a deadlock is detected.

**Procedure** insert\_distributed\_dependency(new\_local\_dependencies:EdgeStack);
**begin**
while not isEmpty(new\_local\_dependencies)
    pop ED from new\_local\_dependencies;
    if ED.T1 > ED.T2 and ED.C2 ≠ LDD; then
        send insert\_edge(ED) to ED.C2
    endif
endwhile
return
end insert\_distributed\_dependency;

**Procedure** remove\_distributed\_dependency(removed\_dependencies:EdgeStack);
**begin**
while not isEmpty(removed\_dependencies) do
    pop ED from removed\_dependencies;
    if ED.T1 > ED.T2 and ED.C2 ≠ LDD; then
        send remove\_edge(ED) to ED.C2
    endif
endwhile;
end remove\_distributed\_dependency;
7.5 An Example of Deadlock Detection

We present an example of a deadlock detection using the algorithm shown in Figure 7.5. Let us assume there are six transactions, \( T_1 \ldots T_6 \), in the system. LDD\(_1\) is the coordinator of \( T_1 \) and \( T_6 \), LDD\(_2\) is the coordinator of \( T_2 \) and \( T_3 \) and LDD\(_3\) is the coordinator of \( T_4 \) and \( T_5 \). Figure 7.3(a) shows dependencies in object A, B, C, D, and E. Object A and B are in ASOS\(_1\) and Object B, C, and D are in ASOS\(_2\). Object A is held by \( T_3 \), object B is held by \( T_4 \), object C is held by \( T_5 \), Object D is held by \( T_2 \), and object E is held by \( T_6 \). Numbers in parentheses represent the sequence of events in the system.

Let the sequence of events be:

1. \( T_6 \) is blocked by \( T_3 \) at object A
2. \( T_2 \) is blocked by \( T_4 \) at object B
3. \( T_2 \) is blocked by \( T_5 \) at object C
4. \( T_1 \) is blocked by \( T_3 \) at object A
5. \( T_5 \) is blocked by \( T_6 \) at object E
6. \( T_3 \) is blocked by \( T_2 \) at object D

The content of each local deadlock detector is shown in figure 7.3(b) before the event 6.

Event 1 : An edge \( <T_6; \text{LDD}_1, T_3; \text{LDD}_2> \) is inserted at \( \text{LDD}_1 \) and a probe \( <T_6; \text{LDD}_1, T_3; \text{LDD}_2> \) is forwarded to \( \text{LDD}_2 \).

Event 2 : An edge \( <T_2; \text{LDD}_2, T_4; \text{LDD}_3> \) is inserted at \( \text{LDD}_2 \).

Event 3 : An edge \( <T_2; \text{LDD}_2, T_5; \text{LDD}_3> \) is inserted at \( \text{LDD}_2 \).

Event 4 : An edge \( <T_1; \text{LDD}_1, T_3; \text{LDD}_2> \) is inserted at \( \text{LDD}_1 \).

Event 5 : An edge \( <T_5; \text{LDD}_3, T_6; \text{LDD}_1> \) is inserted at \( \text{LDD}_3 \); the edge is forward by \( \text{LDD}_1 \) on behalf of object E.

Event 6.1 : An edge \( <T_3; \text{LDD}_2, T_2; \text{LDD}_2> \) is inserted at \( \text{LDD}_2 \) and probes \( <T_3; \text{LDD}_2, T_4; \text{LDD}_3> \), \( <T_3; \text{LDD}_2, T_5; \text{LDD}_3> \), \( <T_6; \text{LDD}_1, T_2; \text{LDD}_2> \), \( <T_6; \text{LDD}_1, T_4; \text{LDD}_3> \), and \( <T_6; \text{LDD}_1, T_5; \text{LDD}_3> \) are created by the local dependency procedure, but only \( <T_6; \text{LDD}_1, T_4; \text{LDD}_3> \) and
\(<T_6;LDD_1,T_5;LDD_3>\) are sent to LDD_3 and the rest are discarded because \(<T_3;LDD_2,T_4;LDD_3>\) and
\(<T_3;LDD_2,T_5;LDD_3>\) are not antagonistic edges and the destination of \(<T_6;LDD_1,T_2;LDD_2>\) is itself. However all six edges are included in TWFG of LDD_2.

**Event 6.2**

- At LDD_3, new local dependencies are calculated for the forwarded edges, \(<T_6;LDD_1,T_4;LDD_2>\) and \(<T_6;LDD_1,T_5;LDD_3>\). A new probe \(<T_6;LDD_1,T_6;LDD_1>\) is created by the local Dependency procedure. The edge \(<T_6;LDD_1,T_6;LDD_1>\) represents one cycle; LDD_3 is detected a deadlock. T_6 is aborted. All edges belonging to T_6 are discarded. Now TWFG of LDD_3 is empty.

**Event 6.3**

- Objects A and E are notified that T_6 is aborted. Object A sends a remove_edge \(<T_6;LDD_1,T_3;LDD_2>\) to LDD_2, and object E sends a remove_edge \(<T_5;LDD_3,T_6;LDD_1>\) to LDD_3 - actually forwarded by LDD_1 on behalf of object E.

**Event 6.4**

- At LDD_2, Anti-probes \(<T_6;LDD_1,T_3;LDD_2>\>,
\(<T_6;LDD_1,T_2;LDD_2>\>, \(<T_6;LDD_1,T_4;LDD_3>\), and
\(<T_6;LDD_1,T_5;LDD_3>\) are created by the local Dependency procedure, but only \(<T_6;LDD_1,T_4;LDD_3>\) and \(<T_6;LDD_1,T_5;LDD_3>\) are sent to LDD_3 and the rest are used to remove corresponding edges(or probes) in TWFG.

**Event 6.5**

- At LDD_3, anti-probes, \(<T_6;LDD_1,T_4;LDD_3>\) and
\(<T_6;LDD_1,T_5;LDD_3>\), are discarded because its TWFG is empty (at event 6.2).

Note that we assume that all deadlock detection messages are delivered. If messages are lost then phantom deadlocks can be found. For instance, if the remove_edge message sent by LDD_1 to LDD_2 at event 6.1 is lost, then LDD_2 will not delete any edges related with T_6 because LDD_2 has no way to know that T_6 is aborted; T_6 never used any object in ASOS_2, so no object reports the event and no other LDDs send a message to notify the event.
Figure 7.5 (a) Transaction dependencies in ASOSs and (b) their representation in local deadlock detectors.
Numbers in parentheses in Figure 7.5(a) and at front of edges in Figure 7.5(b) represent the corresponding events, and prefix P represents probes created by insert_distributed_dependency procedure and prefix F means messages forwarded by local deadlock detectors to coordinators of transactions on behalf of objects.

7.6 The Cost of Deadlock Detection and Resolution

The cost of deadlock detection can be considered as three factors [Sihna and Natarajan 85-1]: communication cost, delay, and storage cost. Communication cost represents the number of messages exchanged to detect deadlocks. Delay is the amount of time between when a deadlock cycle is formed and when the cycle is detected. Storage cost is the amount of space required for deadlock detection.

In distributed deadlock detection, communication between two agents is done via a computer network in which operations are relatively slow and expensive. In our algorithm, the communication cost is reduced by reducing the number of agents involved in deadlock detection. By minimising the number of agents to detect deadlock cycles, delay is also reduced - less messages are forwarded and the total number of transmissions required is half of these of Sihna and Natarajan [85-1]'s and Roesler and Burkhard's [88] algorithms. Delay is related to the length of a deadlock cycle. The storage cost is related to the number of edges in the transaction-wait-for graph. The number of edges representing direct wait-for relationships increases by one when a new direct edge is inserted but the number of edges representing indirect wait-for relationships increases by up to total number of edges which is reachable from the origin of the inserted edge. The transaction managers and objects managers of Sihna and Natarajan [85-1] and Roesler and Burkhard[88] are combined; it effectively reduces the total amount of storage required by half.
Chapter VIII
Conclusions

8.1 Summaries and Conclusions

In this thesis we have proposed an object-space model and a cache coherence protocol for transactions on distributed objects. The protocol includes concurrency control and cooperates with a recovery scheme.

8.1.1 The object-space model

The object-space model is an enhanced version of the client-server model. In the client-server model, servers provide services for clients and clients use services via remote requests. This model provides a good paradigm for sharing services between nodes in local area network. However the client-server model under-utilises client machines which now provide a far better performance than they did at the time when the client-server model was introduced.

In addition, accessing servers is more expensive than accessing local information within a client machine. Reducing remote access is essential for improving the efficiency of the whole system. The proposed object space model uses client machines more heavily than the client-server model. It utilises client machines by distributing some traditional server's tasks to client machines and by reducing the proportion of remote references.

We divide servers into two layers: the active and the passive object layers. The active object layer provides a sharing facility for users and the passive object layer provides persistency for services. Each layer consists of object spaces, called active/passive shared object spaces (ASOSs and PSOSs). Services in a PSOS are highly replicated into ASOSs. The proposed model is designed to achieve three objectives.
• It provides a suitable environment for objects. Unlike an ordinary datum, an object is a structured entity. To deal with objects, more sophisticated methods are required such as flattening objects, tracing objects' relations, providing address spaces within which to run and to store them, etc.

• The model distributes the workload of conventional servers. When a client uses a shared service in a server, its processing speed is likely to depend on that of the server. Conventional servers provide both a storage facility and a sharing facility. In contrast, the division in our model enhances a system's expandability and flexibility. It also reduces communication traffic by moving services nearer to clients. Moreover, the effects of local system failures are reduced significantly.

• The model supports independent service environments, whereas the environment of a conventional server constrains the independence of a service. In our model, each service can use a different naming and concurrency control protocol. It can even be very specialised without interfering with other services by occupying an entire ASOS.

A major drawback of the current model is that the PSOS is not scalable. We discuss this in the later section on future work.

8.1.2 The cache coherence protocol

The cache coherence protocol is proposed to maintain consistency between replicas of a service in context of transactions. The major task of the protocol is to update replicas in the caches as soon as possible with minimum overhead to the system. All transactions are supposed to use the most recent versions of objects and the cache coherence protocol provides such objects. However, due to communication delay, message loss, and concurrent execution, some transactions may violate this requirement. Such transactions are aborted by the protocol if the violation cannot be undone.

When a transaction is aborted, the protocol restores the state of objects affected by the aborted transaction to the most recent consistent state rather than the state with which the aborted transaction started. When a transaction commits, the protocol provides update information to replicas
of services in ASOSs and PSOSs. Each replica in an ASOS may update its version as soon as possible with the supplied information or defer the update until an appropriate time. An update action makes a replica consistent and is equivalent with partial validation of active transactions which use the replica. When an active transaction using the replica is completed, the transaction does not require a separate validation phase unless the current version of replica is out-dated.

Because sending such update information is a by-product of the two-phase commitment protocol using multicast, it avoids unnecessary communication with managers of PSOSs to update active replicas and to validate transactions using these replicas. In addition, the abortion rate of transactions can be reduced by updating replicas early and by considering operation sequences on objects. Active transactions are supplied with up-to-date objects as soon as possible, which reduces the risk of accessing out-dated objects.

When a transaction accesses an out-dated state of an object due to communication delay, the protocol considers the operation sequence on the object to avoid unnecessary abortion of some transactions. This update procedure is equivalent of the validation phase of optimistic concurrency control for a transaction. Unlike the validation phase, a transaction may be involved in more than one update procedure and this is done during execution rather than at the end of the execution of the transaction.

Updating is done in ASOSs: concurrency control and the task of keeping the cache consistent is well distributed into active object domains. Moreover, the protocol permits the usage of local concurrency control. Within a replica of a service, a conflict-based concurrency control protocol takes care of concurrent accesses by multiple transactions. Concurrency control for a service having a single replica works like a conventional conflict-based protocol. Between replicas of a service, the cache coherence protocol controls concurrent accesses.

A comparison of our cache coherence protocol with other work

The proposed protocol has similarities with distributed shared memory (DSM) and (backward and forward) validation in optimistic concurrency control.
The protocol maintain consistency between replicas in caches (ASOSs) as in DSM. However the proposed protocol does not synchronise consistency as in DSM - consistency between replicas in ASOSs is weak. Consistency between replicas is not forced nor guaranteed. This property provides flexible object management environment in ASOSs and makes it possible to use unreliable communication. The proposed protocol tolerates common communication problems such as message delays, message loss, and out-of-order arrivals of messages. The protocol also provides transactions and local concurrency control.

The proposed protocol uses optimistic concurrency control philosophy to maintain consistency. It adopts distributed backward optimistic concurrency control to maintain consistency between a replica of a service in an ASOS and its corresponding service in a PSOS, and distributed forward optimistic concurrency control between replicas of a service in ASOSs. The backward property is used to overcome problems due to unreliable communication and the forward property is used to reduce the abortion rate of transactions and to reduce communication overhead in a local area network. Unlike both optimistic concurrency control protocols, the proposed protocol provides local concurrency control and uses histories of transactions to reduce transaction abortion rate.

To summarise the advantages of our cache coherence protocol are

- it supports a high degree of cache replication of services,
- it distributes cache consistency (and concurrency control) into caches with low communication overhead, and
- it provides a very flexible way to use conventional concurrency control protocols, such as two-phase locking, timestamping and optimistic concurrency control.

8.1.3. Recovery scheme

A recovery scheme is proposed for use with the cache coherence protocol in the object-space model. It combines recovery information and version information to save storage space in repositories (PSOSs). Version information is stored as part of recovery information and it is used to minimise the cost of updating replicas in caches (ASOSs) which are outdated due to message delay or loss. With version information, out-dated
replicas can easily be brought up-to-date rather than reloading these replicas again from PSOSs, which reduces the amount of data to be transferred between caches and repositories. Because repositories do not process version information apart from storing it persistently and version information can be supplied to replicas directly, the response time of repositories is improved.

An object in an ASOS keeps logs of transactions using the object in the ASOS and sends the log of a transaction to PSOSs only when the transaction tries to commit. Therefore, the recovery scheme of PSOSs is simpler than conventional logging schemes; the recovery action consists of removing a single incompletely appended entry from the differential file and it does not require restoration of old values. However, like logs, objects in ASOSs are permitted to update as soon as their state is changed.

In differential file schemes, information of changed states of objects is stored in an addition and deletion file. In our scheme, each version of a group of objects in a PSOS has its own addition and deletion entries and there is not a global addition and deletion file. As mentioned above, these individual addition and deletion file act as a version of the group of objects. Accessing a latest state of an object may take considerable time due to referencing several files, i.e., base, add and deletion files. To avoid this problem, we maintain a shadow copy of the most recent version of the group. The shadow copy may provide a good backup copy of the differential file when the differential file is unrecoverable due to serious hardware failure.

Unless careful replacement is used, the shadow copy will require reconstruction in the event that the PSOS crashes during the updating of the shadow copy.

8.1.4 Distributed deadlock detection

Although replication of a service reduces the probability of deadlocks arising, transactions may still cause deadlocks if replicas employ two-phase locking as their local concurrency control protocol. To detect such deadlocks, we provide a deadlock detection algorithm, which is a modified version of Sihna and Natarajan[85-1]'s and Roesler and Burkhard[88]'s algorithms. The algorithm reduces the number of agents involved in deadlock detection relative to that of Sihna and Natarajan[85-1], and
Roesler and Burkhard[88]. It has several advantages over Sihna and Natarajan’s and Roesler and Burkhard’s algorithms:

- it reduces the number of deadlock detection messages by up to half those generated by other algorithms,
- it reduces the amount of storage required, and
- it is independent of the concurrency control protocol employed by objects.

The drawback of our algorithm is that it may detect phantom deadlocks. If a transaction which is involved in a deadlock cycle is aborted, after the deadlock cycle is detected and before the deadlock cycle is broken. Unnecessary abortion of the transaction which has the highest transaction identifier among transactions in the deadlock occurs. To avoid this problem, the local deadlock detector which detected the deadlock must contact other local detectors to test whether any transaction involved in the deadlock is aborted after the deadlock detection request was made. This is required whenever a deadlock is detected. All local detectors involved in the deadlock are known.

8.2 Future Work

8.2.1 Evaluation and Case Study

In this thesis, the evaluation of the performance is not done, even though we discussed the correctness issue. We need performance evaluation and case studies to demonstrate suitable applications for our object model.

8.2.2 Distributed repositories

In our object space model, we provide cache and repository replication. However we assume that a repository contains all the services in a local area network. This could cause a severe bottleneck and limits the scalability of the model. A set of nodes (computers) can make up a repository; services in the repository can be partitioned between several nodes. In this scheme, services would not be partitioned. We call this distributed repository replication.

Our model can be extended to handle this kind of repository. The distributed repository replication has several advantages.
A distributed repository can be more robust than an undistributed one. When a node is failed, only a part of the repository is inaccessible, but the remaining part of the repository in other nodes can still accessible.

Workload of a repository service is distributed and repository bottleneck problems can be lessened.

Commitment deadlock may happen if more than one partition of a repository is involved at the transaction commitment phase. Commitment deadlock is prevented in our protocol. However, because the number of repository partitions in a system is relatively small and deadlock does not occur frequently, detecting deadlock may be more attractive than preventing it. We can achieve this by using our deadlock detection algorithm. But the cache coherence protocol requires a minor modification to accommodate the new detection method.

8.2.3 Modifying operations

We discussed updating the state of objects in this thesis, but operations of an object can also be changed. This problem requires careful consideration of the inter-changeability between an old version and a new version. Because software requires continuous maintenance and upgrade, this problem must be addressed.

8.2.4 Persistent caches

Non-volatile RAM memory technology has developed rapidly in recent years and computers having this kind of memory have appeared in the market. This kind of memory provides a good hardware support to long term caching; once objects are loaded into a cache which is made non-volatile memory, these objects can survive a very long time even after power off/power failure as well as being able to survive normal system crashes.

These objects can be made reusable by updating them to consistent and most recent versions rather then reloading them every time when they are required. With a suitable architecture, such a recycling of objects can be more effective in term of communication cost. Our architecture can be applied for this purpose.
References


