Design of an object-oriented framework for atomic transactions

Bedir Tekinerdogan

M.Sc. Thesis at the Department of Computer Science

March 1994
Design of an object-oriented framework for atomic transactions

by

Bedir Tekinerdogan

THESIS

Submitted to the University of Twente on March 3, 1994, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Graduation committee:
Dr. ir. M. Aksit (supervisor)
Ir. L.M.J. Bergmans
Ir. J. Bosch

University of Twente
Department of Computer Science
P.O. Box 217
7500 AE Enschede
The Netherlands
To my parents
Contents

Preface 1

Abstract 2

Chapter 1 Introduction 3

1.1 Distributed systems 4
1.2 Transactions 5
1.3 Problem statement 7
1.4 Thesis organization 8

Chapter 2 Atomic Transactions 9

2.1 Concurrency control 10
  2.1.1 Introduction 10
  2.1.2 Concurrency control models 11
  2.1.3 Serializability 12
  2.1.4 Locking Mechanisms 15
    2.1.4.1 Two phase locking 15
2.1.4.2 Distributed Two-Phase Locking 16
2.1.4.3 Deadlock 16
2.1.4.4 Multiple-granularity locking 18
2.1.4.5 Evaluation of Locking mechanisms 20
2.1.5 Timestamp Ordering method 21
  2.1.5.1 Timestamp management 21
  2.1.5.2 Distributed Timestamp ordering 22
  2.1.5.3 Evaluation of the Timestamp Ordering method 22
2.1.6 Optimistic concurrency control 23
  2.1.6.1 Distributed optimistic concurrency control 23
  2.1.6.2 Evaluation of optimistic concurrency control 24
2.2 Recovery 25
  2.2.1 Introduction 25
  2.2.2 Transaction failures 26
  2.2.3 System failures 28
  2.2.4 Media failures 30
  2.2.5 The Commit protocol 30
2.3 Correct executions 33

Chapter 3  Object-oriented analysis and design 36

3.1 Introduction 37
3.2 Object-oriented concepts 38
3.3 Object oriented software development 40
3.4 Problems in the state-of-the-art methods 43
  3.4.1 Problems related to the preparatory work 44
  3.4.2 Problems related to the structural relations 45
  3.4.3 Problems related to object interactions 46
3.5 The Composition-filters model 47
  3.5.1 The basic object model 48
  3.5.2 The Interface part 48
  3.5.3 The implementation part 50
  3.5.4 Inheritance and delegation 51
  3.5.5 Concurrency through input filters 53
  3.5.6 Synchronization through input filters 54
  3.5.7 Abstract Communication Types 55
  3.5.8 Multiple Views through filters 57
  3.5.9 Associative inheritance through filters 58
Chapter 4  Design of the framework  61

4.1  Introduction 62
4.2  Requirements 62
4.3  Decomposition of the transaction system 63
4.4  Introducing dynamic behavior 67

Chapter 5  Implementation of the framework  71

5.1  Introduction 72
5.2  Class ApplicationWithTransaction 73
5.3  Class TransactionManager 74
5.4  Class PolicyManager 75
5.5  Class DataManager 78
5.6  Class UniversalScheduler 80
5.7  Locking scheduler classes 83
5.8  Timestamp ordering scheduler classes 85
5.9  Optimistic scheduler classes 86
5.10  Class SerialScheduler 87
5.11  Class RecoveryManager 88

Conclusion and evaluation  89
Appendix  91
References  93
Index  97
Preface

This thesis presents the results of my assignment which I fulfilled within the Trese-group at the Department of Computer Science, University of Twente in the Netherlands. Trese stands for “Twente Research and Education on Software Engineering”. Within this project problems related with the construction of large and complex software systems are handled. The software development is mainly based on object-oriented concepts. Based on various pilot studies the Trese-group developed the composition filters model. The composition filters model is a modular extension to the current object-oriented model. It can cope with many problems identified in current object-oriented software methodologies. The composition filters model is expressed in the object-oriented programming language Sina\(^1\).

The aim of my assignment was the design (and implementation) of an object-oriented framework for atomic transactions using the composition filters model. As a preliminary work I had to do some study about object-oriented distributed systems. In addition I studied transaction systems in special to gain the necessary knowledge about this problem domain to construct the object-oriented framework. Since during my graduation time the compiler for the Sina language was not totally implemented yet, I had to implement the framework in Smalltalk also.

I think my graduation time was the most instructive period of my whole study. Although it was most of the times real hard work I enjoyed it very well.

I would like to thank my supervisor M. Aksit for his encouragement and advises during my graduation. I am also grateful to my two other supervisors L.M.J. Bergmans and J. Bosch for their support and recommendations to my work.

Bedir Tekinerdogan,

Enschede, 3 March 1994

---

1. Named after Ibni Sina (980-1037) whose contributions to medicine has been used up to the present century.
This thesis describes the design of an object-oriented framework for atomic transactions. Atomic transactions allow their users to assume that their programs are executed atomically as if there were no concurrency and reliable as if there were no failures. So it is an abstraction of an atomic and reliable execution of a program. Concurrency control algorithms are used to synchronize accesses of many users to the same data thereby avoiding inconsistencies. Recovery techniques are used to cope with failures. Several concurrency control algorithms based on the serializability theory will be described in this thesis. An execution is serializable if it produces the same effect as a serial execution. Serializable executions can be provided by ordering conflicting operations in the same way as they would appear in a serial execution.

Large and complex systems have enforced in particularly the requirements of reusability, extensibility and encapsulation to provide maintainable and robust software. The traditional software design methodologies could hardly fulfil these requirements. Object-oriented software development methodologies have proven to be better suitable for the analysis and design of large systems. However, there are still some obstacles which are not addressed by the current object-oriented software development methodologies. The Trese group at the University of Twente in the Netherlands developed the composition filters model that can cope with most of these obstacles. Our framework will also be based on the composition filters model.

The framework is developed to solve the obstacles found in traditional transaction systems. The traditional transaction systems are written as one big program which is difficult to extend. Furthermore, they adopt single transaction semantics whereas different applications may require different transaction semantics. Moreover, they are not flexible enough to switch to different transaction semantics although this may be strongly required by the application, the environment or even the data. This thesis will describe the design of a framework which will contain the constructs to develop a transaction system that can deal with these problems.
Chapter 1

Introduction

Abstract

This chapter describes the basic definitions that are related to atomic transactions in distributed systems. In addition a global model of a transaction system will be presented. Also the problem statement will be described.
1.1 Distributed systems

A distributed system is a system with many processing elements and many storage devices, connected together by a network [Mullender 89]. Potentially, this makes a distributed system more powerful than a centralized system in two ways. First, it can be more reliable, because every function is replicated several times. When one processor fails, another can take over the work. Each data object can be stored on several disks, so that a disk crash does not destroy any information. Second, a distributed system can do more work in the same amount of time, because many computations can be carried out in parallel. These two properties, fault tolerance and parallelism, give a distributed system the potential to be much more powerful than a centralized operating system.

The main goal for the designer of a distributed system is to make the effect of a distribution transparent to the user; users should have a view of the system as a whole. They should not normally be aware of the locations of hardware and software components from which the system is constructed. Thus, the user should view the system as an ordinary centralized operating system, although the system runs on multiple independent CPUs.

We can identify eight forms of transparency [Coulouris 88]:

- **Access transparency** enables local and remote data objects and other objects to be accessed using identical operations.
- **Location transparency** enables objects to be accessed without knowledge of their location.
- **Concurrency transparency** enables several users or application programs to operate concurrently on shared data without interference between them.
- **Replication transparency** enables multiple instances of data to be used to increase reliability and performance without knowledge of the replicas by users or application programs.
- **Failure transparency** enables the concealment of faults, allowing users and application programs to complete their tasks despite the failure of hardware or software components.
- **Migration transparency** allows the movement of objects within a system without affecting the operation of users or application programs.
- **Performance transparency** allows the system to be reconfigured to improve performance as loads vary.
- **Scaling transparency** allows the system and applications to expand in scale without change to the system structure or the application algorithms.

In addition to transparency a distributed system should execute tasks for its users consistently and effectively. There are two main factors that threaten the consistency of data: concurrent executions and failure of processes and computers [Bernstein 87]. To achieve data consistency distributed systems should include provision for both concurrent updating of databases and recovery from system failures.
Since these mechanisms are general requirements, they must be provided by the operating system instead of realizing them in application programs [Aksit 87]. The implementation of these concurrency- and recovery mechanisms, however, should be transparent to the application program developers, since they will use only the primitives and they don’t wish to be bothered with implementation details.

Systems that solve the concurrency control and recovery problem allow their users to assume that each of their programs execute atomically, as if no other programs were executing concurrently, and reliable, as if there were no failures. This abstraction of an atomic and reliable execution of a program is called a transaction.

1.2 Atomic transactions

Informally, a transaction is an execution of a program that accesses a shared database. A transaction includes transaction operations and database operations. Transaction operations are: StartTransaction, Commit and Abort. A program tells the database system that it is about to begin executing a new transaction by issuing the operation StartTransaction. A transaction terminates by either executing a Commit or an Abort operation. A Commit operation implies that the transaction was successful and hence all its updates should be made permanent in the database. By issuing an Abort, the transaction mechanism tells the database system that the transaction has failed, and hence requires the database system to cancel all of its effects. A transaction can also be forced to abort by the system, for instance due to the constraints of concurrency control.

Formally, a transaction is a program that satisfies the following four conditions, which are also known as the ACIDity properties [Haerder 83]:

1. **Atomicity**, or the all-or-nothing property, refers to the fact that all the operations of a transaction must be treated as a single unit; hence, either all the operations are executed, or none.

2. **Consistency** requires a transaction to be correct, i.e., if executed alone, the transaction takes the database from one consistent state to another. When transactions are executed concurrently, the database management system must ensure that the executions of a set of concurrent and correct transactions also maintains the consistency of the database.

The meaning of consistency can be different for every data item. Consistency can be maintained if every data item satisfies the application-specific consistency constraints, meaning that the data item satisfies these constraints before and after the execution of a transaction. For example, in an airline system one consistency constraint might be that each seat on a flight can be reserved by only one passenger.
Chapter 1  Introduction

3. **Isolation** requires each transaction to observe a consistent database, i.e., not to read the intermediate results of other transactions.

4. **Durability** requires the results of a committed transaction to be made permanent in the database in spite of failures. Sometimes, durability is referred to as permanence.

The ACIDity properties of transactions are usually ensured using two different sets of algorithms or protocols. These protocols are divided into ones that ensure execution atomicity, and those that ensure failure atomicity. Execution atomicity refers to the problem of ensuring the overall consistency of the database, and hence the consistency property of transactions, even when they are executed concurrently. Protocols that ensure execution atomicity are called concurrency control protocols. Failure atomicity, on the other hand, ensures the all-or-nothing as well as isolation and durability properties. Protocols that ensure failure atomicity are usually referred to as recovery protocols [Elmagarmid 91].

We assume that each transaction is self-contained, meaning that it performs its computation without any direct communication with other transactions. Transactions only communicate indirectly, by storing and retrieving the states of the data objects in the database. However, this is the only way they can affect each other’s execution.

In the study of concurrency control and recovery we need a model of the internal structure of a transaction system. Figure 1 shows the model of a transaction system. A user interacts with the system in terms of transactions. Database and transaction operations issued by a transaction to the DBS are first received by the transaction manager TM. The task of a TM is to supervise the processing of transactions initiated at a site, while the task of a data manager DM is to manage the access to the actual stored data at a given site. The Data Manager contains a Scheduler and a Recovery Manager (RM). The scheduler controls the relative order in which database and transaction operations are executed. The recovery manager is responsible for transaction commitment and abortion. The RM has also direct access to the object in question.
1.3 Problem statement

In this thesis I shall describe the design of an object-oriented distributed atomic transactions framework. A framework is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuse at a larger granularity than classes [Johnson 88]. The interfaces between the components in a framework are defined in terms of sets of messages. There will usually be a library of subclasses that can be used as components in the design.

Frameworks are more than well written class libraries. In class libraries each component can serve as a discrete, stand-alone, context independent part of a solution to a large range of different problems. Such components are largely application independent. A framework, on the other hand, is an abstract design for a particular kind of application, and usually consists of a number of classes. These classes can be taken from a class library, or can be application-specific. One important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user’s application code. The framework often plays the role of the main program in coordinating and sequencing application activity. This inversion of control gives frameworks the power to serve as extensible skeletons. The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application.

This thesis will describe the design of an object-oriented framework for atomic transactions in both centralized as in distributed systems. We assume that data objects are not replicated, that is, each data object is stored at a single site.

The framework has to be developed to solve the obstacles found in traditional transaction systems. The
traditional transaction systems are written as one big program which is difficult to extend. Furthermore, they adopt single transaction semantics whereas different applications may require different transaction semantics. Moreover, they are not flexible enough to switch to different transaction semantics although this may be strongly required by the application, the environment or even the data. This thesis will describe the design of a framework which will contain the constructs to develop a transaction system that can deal with these problems.

We shall adopt the composition filters model to design the framework.

1.4 Thesis organization

The remainder of this thesis is organized as follows:

Chapter 2 provides the description of atomic transactions. The main subjects here will be concurrency control and recovery in both centralized as in distributed systems. The concurrency control model used here is based on the serializability theory.

Chapter 3 will describe object-oriented analysis and design. First basic concepts of the object-oriented approach will be presented. This will be followed by an evaluation of current object-oriented software development methodologies. The chapter will be concluded by a presentation of the composition filters approach that presents solutions to most of the problems noticed in current object-oriented software methodologies.

Chapter 4 describes the design of an object-oriented framework for atomic transactions. The design is not based on any rigorous design methodology. By decomposing the transaction system in well-defined subcomponents the design will be explained.

Chapter 5 presents the implementation of the framework.

The thesis concludes with the conclusions and evaluation of the framework. Appendix A contains the Sina implementation of the framework. Appendix B contains the Smalltalk sources. These are included in a supplementary report.
Chapter 2

Atomic Transactions

Abstract

Atomic transactions are needed to maintain the consistency of data and to cope with failures. Concurrency control algorithms are used to synchronize accesses of many users to the same data thereby avoiding inconsistencies. Several concurrency control algorithms based on the serializability theory will be described here. Recovery techniques used for dealing with failures will be presented. Finally, a notion of correct executions will be given. These are executions that provide both serializable as at least recoverable executions.
2.1 Concurrency control

2.1.1 Introduction

When transactions run concurrently, there are many different possible interleavings of their read and write operations and, if they access the same data items, their effects are unpredictable. Consider for example a bank account s, and two transactions, each withdrawing $100 when the account has an initial balance of $300. We assume that a withdraw transaction is executed by first reading the value of account s, and then writing it with the new value. Assume that the interleaved execution of the two withdraw transactions is as shown in figure 2.1.

<table>
<thead>
<tr>
<th>Transaction T1</th>
<th>Transaction T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>withdraw($100)</td>
<td>withdraw($100)</td>
</tr>
<tr>
<td>Read(s)</td>
<td>Read(s)</td>
</tr>
<tr>
<td>Write(s, s-100)</td>
<td>Write(s, s-100)</td>
</tr>
</tbody>
</table>

Figure 2.1 Lost Update

In this execution, both transactions read the same initial value s, and then alter it. As transaction T2 writes its result after transaction T1, the result is incorrect because the balance of account s is decreased by $100 instead of $200. This is an illustration of the lost update problem. Consider another type of transaction which transfers $100 from a savings account to a checking account c. Assume that both transactions start with an initial balance of $200. Further, assume that a transfer transaction was executed concurrently with a balance transaction that computes the total amount in both accounts as shown in figure 2.2.

In this execution the balance operation returns a sum of $300, which is incorrect since the two accounts actually have $400. Again note that if the two transactions were executed one at a time (in either order), the balance transaction would have returned the correct total of $400. The problem here has to do with the balance transaction observing an inconsistent state. This problem is referred to as the inconsistent retrieval.
Chapter 2  Atomic Transactions

2.1.2 Concurrency control models

The examples above illustrate that simply requiring each transaction to be individually correct does not guarantee that the interleaved execution of a set of transactions will be correct. One way to avoid interference problems is not to allow transactions to be interleaved at all. An execution in which no two transactions are interleaved is called serial [Papadimitriou 86]. More precisely, an execution is serial if, for every pair of transactions, all of the operations of one transaction execute before any of the operations of the other. Requiring the transactions to be serial means total elimination of concurrency which would degrade such important aspects of the performance of the system as response time and transaction throughput. Moreover, if a pair of transactions do not share a common data item, it is clear that interleaving their operations can not violate any integrity constraints. Such transactions can be executed concurrently, resulting in a better CPU utilization. Furthermore, even if a pair of transactions share some data items, interleaving the execution of their operations may be acceptable.

A simple solution is to run transactions concurrently if they are using different data objects and in series if they are using the same data objects. This solution is not altogether satisfactory for:

(1) The set of database operations composing a transaction can be unknown at the moment of its initialization, but determined dynamically during its execution. As a result, it is then not possible to predict which objects will be eventually used by a transaction. This situation occurs if a transaction contains, for example, a statement such as:
   "if x > 0 then update y, else update z"
(2) A pair of transactions may use the same item for only a small fraction of their time.

It is clear that a concurrent execution of a set of transactions requires control of the way in which database operations originating from different transactions interleave. This control, called concurrency control, consists of a schedule of the database operations. Any schedule (sometimes called history in the literature) is a result of the application of a concurrency control algorithm to the set of transactions. The size of the set of different correct schedules generated by a concurrency control algorithm is a measure of the concurrency degree it provides [Cellary 89]. In addition it is a function of the information that it has at its disposal: information about the structure and organization of the database system, about the consistency constraints imposed on the database system about the set of database operations involved in transactions, about the computations performed by transactions, etc. The scope and character of the information available for concurrency control algorithms determines the concurrency control model. Concurrency control models that have only information about the sets of data objects whose access is required are called syntactic. Semantic based concurrency control models have semantic information at their disposal. This information can concern data (e.g. physical structure of the database, consistency constraints) or the set of transactions. More about semantic based concurrency control can be found in [Garcia-Molina 83] and [Lynch 83]. The syntactic concurrency control model is the one which produces serializable executions. This is the concurrency control model that will be presented in this chapter.

2.1.3 Serializability

An execution is serializable if it produces the same output and has the same effect on the database as some serial execution of the same transactions. Since serial executions are correct, and since each serializable execution has the same effect as a serial execution, serializable executions are correct too [Bernstein 87]. We can provide serializable executions by ordering conflicting operations of non-aborted transactions in the same way as they would appear in a serial execution. Two operations conflict if both of them access the same data object and one of them is a write operation [Cellary 89] (see figure 2.3).

![Conflict relation read and write operations](image_url)
Consider for example the following situation. Let us assume that three accounts X, Y, Z have the balances 200, 400 and 600 dollars, respectively. Consider the standard example with two transactions which transfer funds from one account to another: T1 transfers 100 dollars from account X to account Y, and T2 transfers 200 dollars from account Y to account Z. Any serial execution of these transactions preserves the value of the sum X+Y+Z. A serial execution (T2, T1) is given in figure 2.4.

\[
\begin{array}{c}
T1 \\
\hline
Read(X) \\
Write(X, X-100) \\
Read(Y) \\
Write(Y, Y+100)
\end{array} \quad \begin{array}{c}
T2 \\
\hline
Read(Y) \\
Write(Y, Y-200) \\
Read(Z) \\
Write(Z, Z+200)
\end{array}
\]

Figure 2.4  A serial execution of transactions T1 and T2

Two different interleaved executions of T1 and T2 are shown in figure 8. The balance of accounts X, Y, Z after both the serial and serializable executions of T1 and T2 are the same: X=100, Y=300, Z=800. In both these executions the sum X+Y+Z is preserved. The non-serializable execution gives the final balances: X=100, Y=500, Z=800. These balances violate the requirement that the value of X+Y+Z be preserved. Since the order of transaction execution is arbitrary, an execution is serializable if it is equivalent to any serial execution of the same transactions. The serializable execution in figure 2.5 is equivalent to the serial execution of T2 followed by T1 (figure 2.4). The conflicting operations here are Write2(Y-200) and Read1(Y). Another serializable execution could be equivalent to the serial execution of T1 followed by T2.
A simple way to recognize serializable executions is to construct a serialization graph. This graph has a node for each transaction and an edge $T_i \rightarrow T_j$ if $T_i$ executes an operation which conflicts and precedes some operation executed by $T_j$. An execution is serializable if and only if its serialization graph is acyclic [Bernstein 87]. The serialization graph of the two executions in figure 2.5 are as in figure 2.6.

### Figure 2.5
Serializable and nonserializable executions of two transactions

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(Y)</td>
<td>Read(X)</td>
<td>Read(X)</td>
<td>Write(X, X-100)</td>
</tr>
<tr>
<td>Read(X)</td>
<td>Write(Y, Y-200)</td>
<td>Read(Y)</td>
<td>Write(Y, Y-200)</td>
</tr>
<tr>
<td>Write(X, X-100)</td>
<td>Read(Z)</td>
<td>Write(Y, Y+100)</td>
<td>Write(Y, Y+100)</td>
</tr>
<tr>
<td>Read(Z)</td>
<td>Write(Z, Z+200)</td>
<td>Read(Z)</td>
<td>Write(Z, Z+200)</td>
</tr>
<tr>
<td>Write(Y, Y+100)</td>
<td>Write(Z, Z+200)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| serializable execution | nonserializable execution |

**Figure 2.6** Serialization graph for 1) serializable execution and 2) nonserializable execution

Note that ensuring local serializability at each object does not guarantee overall serializability. Consider for example a distributed database with two objects O1 and O2. Two transactions T1 and T2 may both write both objects and due to the asynchrony of the system, T1 may be serialized before T2 on object O1 and vice-versa on object O2. Each object has a correct serialization order, but the overall execution is not correct. Hence, concurrency control protocols must synchronize the global execution of transactions.

Several efficient concurrency control protocols based on the serializability theory have been proposed.
including two-phase locking [Eswaran 76], timestamp ordering [Bernstein 81] and optimistic concurrency control [Kung 81].

2.1.4 Locking Mechanisms

Locking is a mechanism commonly used to solve the problem of a synchronizing access to shared data. The idea behind locking is intuitively simple. Each data item has a lock associated with it. It is assumed that each transaction before performing a read or write operation on a data item must lock it, and that each transaction will release all locks it holds before its completion. A transaction locks data items to ensure their inaccessibility for other transactions during the period when the database is inconsistent. There are two basic lock modes: read locks and write locks. Two locks are compatible if they can be applied to the same data item by two or more different transactions. Otherwise, locks are incompatible or in other words locks conflict [Date 83]. Lock mode compatibility for the classic locking method is shown in figure 2.7. The compatibility of lock modes follows from their semantics. Two or more transactions can simultaneously read a data item. However, when a transaction updates a data item, i.e. writes, then other transactions cannot either read or write it.

<table>
<thead>
<tr>
<th>requested lock</th>
<th>readlock</th>
<th>writelock</th>
</tr>
</thead>
<tbody>
<tr>
<td>readlock</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>writelock</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Figure 2.7  The compatibility of lock modes

2.1.4.1 Two phase locking

The most used algorithm of the locking method is two-phase locking. The mechanism depends on well-formed transactions, which do not relock entities that have been locked earlier in the transaction and are divided into a growing phase in which locks are only acquired and a shrinking phase, in which locks
are only released. During the shrinking phase, a transaction is prohibited from acquiring locks. If a transaction tries during its growing phase to acquire a lock that has already been acquired by another transaction, it is forced to wait. This situation might result in deadlock if transactions are mutually waiting for each other’s resources.

2.1.4.2 Distributed Two Phase Locking

Two phase locking can also be used in a distributed environment. Since each data object has all the information it needs to decide whether to admit a lock for a transaction, locking can be done without communication with other data objects. Somewhat more problematic is the issue of when to release a lock. According to the 2PL protocol an object O₁ may not release a lock for a transaction Ti until it knows that transaction Ti will not require any more locks on object O₁ or any other object On. Otherwise, an object might release Ti’s lock and some time later another object might be locked for Ti, thereby violating the rule for the 2PL protocol. It would appear that enforcing the 2PL rule requires communication among the objects. However, if transactions read and write only to committed data, then they can avoid such communication, because at the time Ti wishes to commit it surely has obtained all the locks it will ever need. Thus, each object knows at receipt of commit or abort that Ti shall not require locks anymore, and that the locks of Ti can be released together.

2.1.4.3 Deadlock

Deadlock is defined as a system state where two or more transactions are mutually waiting for each other to release data objects necessary for their completion. Deadlock can be local or distributed. In a local deadlock two transactions are waiting for each other until the other transaction releases a conflicting lock on the same object. This can occur for instance when two transactions are waiting to set a writelock on an object after they have both readlocked the same object (see figure 2.8).

In a distributed deadlock transactions are mutually waiting for each other to release different objects. (see figure 2.9).
There are different mechanisms for deadlock resolution. We can classify them into three categories.

(1) **Time-out.**
One of the simplest ways to resolve deadlock is to specify a maximum wait time, and abort the transaction which is waiting if the time expires before the request is granted. The disadvantage of this strategy is that the system may abort a transaction that isn’t really involved in a deadlock but is just waiting for a lock owned by another transaction that is taking a long time to finish. There is no harm done by making such an incorrect step, insofar as correctness is concerned. There is certainly a performance penalty to the transaction that was unfairly aborted.

(2) **Deadlock prevention**
Deadlock prevention is a scheme in which a test is applied at the moment when a transaction Ti requests incompatible access to a data object previously assigned to a transaction Tj to see if there is danger of deadlock or not. If the test is negative one of the transactions Ti or Tj is aborted.

The two best known procedures for deadlock prevention are the Wait/Die and Wound/Wait protocols
which use timestamps to perform the test [Cellary 89].

**Wait/Die:** if \( ts(T_i) < ts(T_j) \) then Ti waits else abort Ti

**Wound/Wait:** if \( ts(T_i) < ts(T_j) \) then abort Tj else Ti waits

The basic philosophy is that an older transaction (one with a smaller timestamp) is favored when conflicting with a younger transaction on the assumption that it may have already used more resources and thus more cost will be paid for abort and restart.

If a transaction has already been aborted then the abort in the Wound/Wait protocol is ineffective in killing the transaction. That is why it’s called wound and not kill. Transaction Tj is wounded, in a (possibly unsuccessful) attempt to kill it.

(3) **Deadlock detection**

Deadlock detection can be done by the construction of a directed graph called a wait-for-graph (WFG). The nodes of the graph represent the waiting transactions, and the directed edges indicate for which transaction a given transaction is waiting. A deadlock occurs if the WFG has a cycle. When cycles are detected, they are broken by choosing victims to be aborted.

The basic difficulty in implementing a mechanism for deadlock detection is the problem of an efficient construction of waits-for graphs for transactions which are distributed over several objects. We distinguish two techniques of the wait-for-graph construction [Bhargava 86]: a centralized technique, where the waits-for-graph is positioned at a selected object, and a distributed technique where fragments of the graph are distributed over several objects. Both techniques require periodic communication between data objects. The centralized technique is inconsistent with the idea of distributed processing, whereas algorithms for the distributed techniques are inefficient. The designer of the detectors must also make some decisions about the detector initiation: is detection made periodically using a predefined time parameter, is it initiated every time a transaction has to wait, or when any suspended transaction has waited for more than a predefined period of time.

More about distributed deadlocks can be found in [Badal 86], [Obermarck 82] and [Knapp 87].

2.1.4.4 **Multiple-granularity locking**

So far we have viewed the database as an unstructured collection of data items. This is a very abstract view. In reality a data item could be a block or page of data, a file, a record of a file, or a field of a record. The granularity of a data item refers to that item’s relative size. For instance, the granularity of a file is coarser, and the granularity of a field finer, than that of a record. The granularity of data items is
unimportant as far as correctness is concerned. The granularity is important, however, when it comes to performance. Multiple granularity concurrency control protocol aims to minimize the number of locks used while accessing sets of objects in a database [Gray 75].

In this model data items are organized in a tree where small items are nested within larger ones. Each nonleaf item represents the data associated with its descendants. The root of the tree represents the whole database. See figure 2.10 for the lock type graph. Each edge in the graph connects a data type of coarser granularity to one of finer granularity. Transactions can lock nodes explicitly, which in turn locks descendants implicitly. For example, a read lock on an area implicitly read locks the files and records in that area. It is also necessary to propagate the effects of fine granule locking activity to the coarse granules that "contain" them. To do this, each lock type has an associated intention lock type. So, in addition to read and write locks, we have intention read locks (irl) and intention write locks (iwl). Before it locks x, the scheduler must ensure that there are no locks on ancestors of x that implicitly lock x in a

![Figure 2.10](image) A lock type graph

<table>
<thead>
<tr>
<th>requested lock mode</th>
<th>R</th>
<th>W</th>
<th>IR</th>
<th>IW</th>
<th>RIW</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>W</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>IR</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>IW</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>RIW</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

![Figure 2.11](image) A compatibility matrix for multigranularity locking
conflicting mode. To accomplish this, it sets intention locks on those ancestors. For example before setting a read lock on record x, it sets ir locks on x’s database, area and file ancestors (in that order). For any y, irl[y] and wl[y] conflict. Thus, by setting irl[y] on every ancestor y of x, the scheduler ensures that there is no wl[y] that implicitly write locks x. For the same reason, iwl[x] conflicts with rl[x] and wl[x] (see figure 2.11).

Suppose a transaction reads every record of a file and writes into a few of those records. Such a transaction needs both a read lock on the file (so it can read all records) and an iw lock (so it can write lock some of them). Since this is a common situation, it is useful to define an riw lock type. An riwl[x] is logically the same as owning both rl[x] and iwl[x]. The compatibility matrix for the five kinds of locks is used to determine when to grant requests and when to deny them. The following multiple granularity protocol based on the compatibility matrix has been proposed [Gray 75]:

1. A transaction Ti can lock a node in rl mode or irl mode only if all ancestors of the node are locked in either iw or ir mode by Ti.
2. A transaction Ti can lock a node in r, riw or iw mode only if all the ancestors of the node are locked in either riw or iw mode by Ti.
3. Locks should be released either at the end of the transaction (in any order) or in leaf-to-root order. In particular, if locks are not held to the end of the transaction, the transaction should not hold a lock on a node after releasing the locks on its ancestors.

The multiple granularity protocol increases concurrency and decreases overhead. This is especially true when there is a combination of short transactions with a few accesses and transactions that last for a long time accessing a large number of objects such as audit transactions that access every item in the database [Elmagarmid 91].

2.1.4.5 Evaluation of Locking mechanisms

The main advantages of the locking method are the following:

- Distribution is possible either by requiring the algorithm to provide strict executions or by inter-object communication.
- The algorithm can be extended by incorporating other locks or strategies

The main disadvantages are:

- Access to data items is restricted because of locks.
- Performance failures like deadlock can occur, whose detection and elimination is very
costly. Another performance failure is permanent blocking also called starvation. This phenomenon occurs when a transaction waits indefinitely for a data access granting because of a steady stream of other transaction whose data access requests are always granted before.

## 2.1.5 Timestamp Ordering method

In timestamp ordering (TO), the transaction manager assigns a unique number, called a time-stamp, \( ts(T_i) \), to each transaction \( T_i \). The timestamps are chosen from a monotonically increasing sequence. This sequence is often a function of the time of day. Using timestamps, a concurrency control mechanism can totally order requests from transactions according to the transactions’ timestamps. The scheduling algorithm is very simple: perform all pairs of conflicting operations in timestamp order. A transaction is aborted if it tries to execute an operation out of timestamp order. That is, if the timestamp of a transaction reading an object is smaller than the timestamp of the latest writer on that object, the transaction is aborted. Similarly, if the timestamp of a transaction writing an object is smaller than the timestamps of the latest reader or writer on that object, the transaction is aborted. Thus, a TO execution has the same effect as a serial execution in which the transactions appear in timestamp order.

There are two ways in implementing the TO method, called the Basic-TO and Conservative-TO [Bernstein 87]. In basic-TO the transaction is restarted after an out of timestamp order, whereas in conservative-TO transactions are delayed until it is sure that a restart will not cause a conflict.

The basic timestamp method does avoid deadlock (simply because no transaction becomes blocked), but it is quite prone to restarts. A modification known as ‘ignore obsolete write’ rule or the Thomas Write rule is an improvement [Coulouris 88]:

If a write is too late, then if it had arrived in time its effects would have been overwritten anyway. Therefore it can just be ignored instead of aborting the transaction. However, if the earlier transaction had read the item, the later one will fail due to the read timestamp on the data.

### 2.1.5.1 Timestamp management

Usually, TMs assign timestamps to transactions. If there is only one TM in the entire system, then it can easily generate timestamps by maintaining a counter. To generate a new timestamp, it simply increments the counter and uses the resulting value. If there are many TMs, as in a distributed environment, then a method is needed to guarantee the total ordering of timestamps generated by different TMs. It is desirable to find a method that doesn’t require the TMs to communicate with each
other, which would make the timestamp generation activity more expensive. The following technique is usually used to make this guarantee. Each TM is assigned a unique number (for example its process or site identifier). In addition, each TM maintains a counter as before, which it increments every time it generates a new timestamp. However, a timestamp is now an ordered pair consisting of the current value of the counter followed by the TMs unique number. The pairs are totally ordered, first by their counter value and second, in case of ties, by their unique TM numbers. The local counter used by each TM can be an actual clock. If a clock is used, then the TM obviously should not increment it to guarantee uniqueness. Instead, it should simply check that the clock has ticked between the assignment of any two timestamps. Since timestamps increase monotonically with time and are unique, if a transaction lives long enough it will eventually have the smallest timestamp (i.e. it will be the oldest) in the system.

2.1.5.2 Distributed Timestamp ordering

The timestamp ordering method is especially easy to distribute. As I noted before, a timestamp is total ordered and it is unique over the entire system. So when we use timestamp ordering at an object to provide concurrency control the operations will actually be ordered in a global manner. Hence, a situation where two transactions are different serialized at two objects cannot occur simply because both objects use the same information to order the operations they receive. Unlike distributed 2PL, where coordination among objects is usually needed to handle distributed deadlocks, distributed TO requires no inter-object communication whatsoever.

2.1.5.3 Evaluation of the Timestamp Ordering method

Advantages

• One transaction can update a data object just after another transaction has read it; data updates can be performed concurrently as long as the timestamp order is not violated. Both these features increase the concurrency degree.
• Deadlock and starvation do not occur.
• Easy to distribute because of the global value of each timestamp in the whole system.

Disadvantages

• As a result of the great pessimism of the algorithm the mechanism will cause a lot of restarts.
• The system must have adequate timestamp generators which must be synchronized in some way. This can be complex and expensive.
2.1.6 Optimistic concurrency control

The basic idea behind the optimistic concurrency control is the assumption that data access conflicts occur very rarely, because most executions are correct and do not require any concurrency control. Therefore, no transaction is aborted or delayed during its execution as it is done by means of locking or timestamp ordering methods. Instead, each transaction is first executed to its commit point and then a validation test is performed to see whether or not there have been any data access conflicts that may have caused consistency of the database to be lost. So scheduling activities are not done for every operation but for a transaction as a whole.

We call the process of checking whether a transaction’s Commit can be safely scheduled or must be rejected a certification. Certification can either be based on two-phase locking or the timestamp method [Bernstein 87]. When the certification is based on 2PL and Ti sends a commit then we must check if there is any operation pi of Ti that conflicts with some operation qj of some other active transaction Tj. If so, transaction Tj will be aborted.

Testing for conflicts during the certification can be done by looking at intersections of the readsets and writesets of the active transactions. The readset of a transaction contains all the objects the transaction has read, the writeset contains all the objects the transaction has written. Assume that readsetTx and writesetTj represent the readset respectively the writeset of transaction Tx. A commit of a transaction Ti may now only be accepted if:

\[
\begin{align*}
\text{readsetTi} \cap \text{writesetTj} &= \emptyset \\
\text{writesetTi} \cap \text{readsetTj} &= \emptyset \\
\text{writesetTi} \cap \text{writesetTj} &= \emptyset
\end{align*}
\]

For each active transaction Tj.

In timestamp ordered certification tests, the operation ci of transaction Ti is only accepted if all conflicts involving operations of Ti are in timestamp order. That is, in the execution produced thus far, if some operation pi(x) precedes some conflicting operation qj(x) of transaction Tj then timestamp(Ti) < timestamp(Tj).

2.1.6.1 Distributed optimistic concurrency control

Although each object certifies a transaction, the activity of transaction certification should be carried out in a coordinated manner in a distributed environment. To certify a transaction a decision must be reached involving all of the certification the objects made for a specific transaction. A global decision
can be reached by consensus. If the local decisions of all the certification tests result in true then the transaction may commit. If even one local certification results in false then the global decision is to abort the transaction. This kind of consensus can be reached by using the two-phase commit protocol between the TM of the transaction that wishes to commit and the objects the transaction accessed. This will be handled in much greater detail in the next section about recovery.

2.1.6.2 Evaluation of optimistic concurrency control

*Advantages*
- The execution of transactions that do not violate database consistency (especially queries) is not delayed.
- Deadlock does not occur.
- Easy to distribute (if two-phase commit protocol provided).

*Disadvantages*
- Information about the transactions has to be collected so that it can be used to perform certification.
- Starvation is possible.
2.2 Recovery

2.2.1 Introduction

Computer systems are subject to many types of failures. Operating systems fail, as does the hardware on which they run. When a failure occurs, one or more programs may be interrupted while they were running. Since the program was written to be correct only under the assumption that it executed in its entirety, an interrupted execution can lead to incorrect results. For example, a money transfer application may be interrupted by a failure after debiting one account but before crediting the other. Avoiding such incorrect results due to failures is called the recovery problem [Bernstein 87].

A recovery algorithm monitors and controls the execution of programs so that the database includes only the results of transactions that run to a normal completion. If a failure occurs while a transaction is executing, and the transaction is unable to finish executing, then the recovery algorithm must wipe out the effects of the partially completed transaction. That is, it must ensure that the database does not reflect the results of such transactions. Moreover, it must ensure that the results of transactions that do execute are never lost.

It is not realistic to expect to build database systems that can tolerate all possible faults. However, a good system must be capable of recovering from the most common types of failures automatically. There are three types of failures that are most important in centralized systems [Bhargava 86]:

- **transaction failure** is when one of the servers participating in a transaction decides to abort the transaction.
- **system failures** (for example CPU failure) that affect all transactions currently in progress but do not damage the database.
- **media failure** (for example disk head crash) that damage the database, or some portion of it, and affect all transactions currently using that portion.

Since distributed systems are more complex than centralized ones they have to deal with additional failures which result from the distribution of the components in the system. A distributed system roughly consists of two kinds of components: sites, which process information, and communication links, which transmit information from site to site. A distributed database is a database where different objects may be stored at different sites and where the users may issue transactions at any site in the system [Ozsu 91], [Bhargava 86]. In a database system, sites are assumed to be fail-stop, that is, the site simply fails by stopping. It never performs incorrect actions. In addition to site failures, communication links may fail to deliver messages. Combinations of both failures may lead to partitioning failures.
[Mullender 89]], where sites in a partition may communicate with each other, but no communication can occur between sites in different partitions.

In distributed systems there are not only additional failures but the nature of failures is also different. In centralized systems a failure is an all-or-nothing property. Either the system is working and transactions are processed routinely, or the system has failed and no transaction can be processed at all. In a distributed system, however, we can have partial failures. Some sites may be working while others have failed.

The site at which a transaction is initiated is called its home site. The home site executes the operations of a transaction by exchanging messages with the sites where the objects are stored. However, in case of a commit, merely having the TM of a distributed transaction’s home site send Commit operations to all other sites is not enough because of possible partial failures in the system. The well known protocol to cope with this situation is called the two-phase commit protocol.

In section 2.2.2, 2.2.3 and 2.2.4 the failures which are inherent to centralized systems will be handled. Section 2.2.5 will describe the two-phase commit protocol which is used to cope with failures in a distributed system.

### 2.2.2 Transaction failures

This section will describe how failure atomicity of transactions can be guaranteed when transactions abort as a result of faults. Note that a user initiated abort can also be modeled as a fault in the system. Consider, for example, the following execution of a transaction:

```
read1[x]write1[x,5]write1[y,25]
```

If in this example the next operation of transaction 1 is `abort1`, due to a fault, then the above operations should be undone. That is, the partial effects of transaction 1 on the database state must be eliminated. Note that there is no need to perform any undo action for read operations since they do not modify the state of the database. On the other hand, write operations must be undone to restore the database to the correct state.

Failure atomicity of transactions in the absence of concurrency can be easily accomplished by using a simple bookkeeping technique. Read operations do not require any undo action whereas write operations require restoration of the before-image of the data objects involved in the write operations.
This can be accomplished by requiring that transactions save the before-image of each object on non-volatile storage before performing a write operation on that object. Before-images are usually stored in a structure called log on non-volatile storage. However, guaranteeing failure atomicity of transactions in a concurrent environment becomes more complex. In this case recovery cannot be achieved by just restoring the altered data objects to the state they were in before the transaction started, for other concurrent transactions may meanwhile have made changes to some of these objects. Consider, for example, the following execution of two transactions:

\[
\text{read}_1[x]\ \text{write}_1[x,5]\ \text{read}_2[x]\ \text{write}_2[z,20]\ \text{commit}_2
\]

In the above execution, if transaction 1 aborts, then we have the following problem. Since the effects of transaction 1 must be eliminated from the database, transaction 2 through its read action \(\text{read}_2[x]\) observes a value of \(x\) that did not exist. The only alternative left now is to withdraw the commitment of transaction 2. However, this approach of guaranteeing failure atomicity violates the property of permanence (durability) that states that once a transaction commits, all its effect become permanent. In order to circumvent this problem, the concurrent executions of transactions is constrained. The constraint requires that if a transaction \(T_1\) reads a value of an object written by transaction \(T_2\) then the commit action of \(T_1\) can be executed only after \(T_2\) has committed. Executions that satisfy this property are referred to as recoverable executions [Hadzilacos 88].

We can now summarize a protocol for guaranteeing failure atomicity. When a transaction aborts, its effects on the database are eliminated by restoring the before-image of the objects written by the transaction, and its effects on other transactions are eliminated by aborting all transactions that read values written by the aborted transaction. Note that this may lead to further abortions of other transactions and so on. This phenomenon is referred to as cascading aborts and can be illustrated with the following example:

\[
\text{write}_1[x,5]\ \text{read}_2[x]\ \text{write}_2[z,20]\ \text{read}_3[z]...
\]

In the above execution, if transaction 1 aborts, it will induce an abort of transaction 2 since the latter read uncommitted values from transaction 1. Furthermore, abortion of transaction 2 will trigger the abort of transaction 3. Although this approach is correct from a theoretical point of view, the phenomenon of cascading aborts is undesirable in practice. This phenomenon is prevented by introducing another property for transactions called isolation. Transactions are isolated from each other by requiring that transactions read only committed data. Executions that follow this constraint are referred to as executions that avoid cascading aborts [Hadzilacos 88].

In this restricted execution model, failure atomicity of transactions can be ensured by simply eliminating the effects of aborted transactions from the database. Note that the issue of aborted
transactions on other concurrent transactions does not arise as a result of the isolation property. However, as the following example illustrates, the simple approach of restoring before-image is not sufficient.

\[ write_1[x,1] write_2[x,2] \]

Suppose the initial value of \( x \) is zero and if transaction 2's commit action is followed by the abortion of transaction 1, restoring the initial value will result in the effects of transaction 2 on \( x \) being lost. (This is analogous to the lost update problem due to transaction aborts instead of transaction commit.) On the other hand, if transaction 1 and 2 abort in that order, restoring the before-image of \( x \) by transaction 2 will result in \( x \) having an incorrect value of 1 instead of zero. In order to use the simple recovery mechanism of restoring the before-images of aborted transactions, an additional constraint is imposed on the concurrent execution of transactions. This new constraint requires that if a transaction \( T \) performs write operation on an object \( x \), then the other transactions are not permitted to write \( x \) until \( T \) has either committed or aborted. This constraint, in conjunction with the constraint for avoiding cascading aborts, can be stated as follows:

If a transaction \( T \) writes an object, then that object can be read or written by other transactions only after \( T \) has either committed or aborted.

Executions that satisfy the above constraint are referred to as strict executions [Hadzilacos 88]. In an execution model that permits only strict executions, an aborted transaction is undone by simply restoring the before-images corresponding to the write operations executed by the aborted transaction. Strict executions greatly simplify the task of ensuring failure atomicity; however they have the expensive of reduced concurrency.

### 2.2.3 System failures

In this section the problem of ensuring failure atomicity of transactions in the presence of system failures is discussed. We use the term "system failure" to mean any event that causes the system to stop and thus requires a subsequent system restart: The contents of main storage are lost but the database is not damaged. Database systems usually use two types of memory, volatile, which is usually small but has fast access time, and nonvolatile, such as a disk, which has slower access time, but has much larger capacity. The entire database is stored in nonvolatile memory, and part of the database is cached or buffered in the volatile memory, usually referred to as the database cache. A transaction usually executes by reading and updating the database cache. Ideally, when a transaction commits, all its changes should be incorporated into the database in nonvolatile storage. Since disk accesses are generally slower, coherency between the database cache and the database itself is not strictly
maintained. Hence, the database may contain data written by uncommitted transactions (which may potentially abort) and the cache may contain data, which has not yet been incorporated into the database, but that was written by committed transactions. A system failure is modeled by assuming that the state of the system in the volatile memory is lost, but the state in the nonvolatile memory survives following the system failure.

The recovery from system failure involves the following [Elmagarmid 91]:
1. The effects of transactions that were in committed state at the time of the system failure must be incorporated into the database.
2. The effects of transactions that were aborted or were active at the time of the system failure must be eliminated from the database. Note that transactions that were active when the failure occurs are deemed to be aborted since their internal state is lost due to the failure.

Recovery techniques can be based on shadowing or write-ahead logging. Shadow recovery goes as follows [Bhargava 86]:
1. When a transaction tries to access a data object, a copy of that object is made, and all work is performed on that copy. The copy is known as a shadow.
2. When a transaction commits, its shadows are installed in favor of the original versions.
3. When a transaction aborts, its shadows are discarded, and the original versions are left intact.

Provided that the installation of shadows can be made to appear atomic, the above rules make the transactions all-or-nothing, guaranteeing failure atomicity. Note that we don’t access the originals directly but in fact we defer the operations until a commit is received. This is called defer-updating. A closely related recovery method is to operate on the originals instead of the shadows and install the shadows on abort, but leave the originals on commit. We call this in-place-updating. Restoring shadows on abort is very similar to logging since in both techniques we use in-place updating.

In recovery techniques based on logging, stable storage contains an append-only sequence of records. Many of these records contain an undo component that permits the effects of aborted transactions to be undone, and a redo component that permits the effects of committed transaction to be redone. Updates to data objects are made by modifying a representation of the object residing in volatile storage and by spooling one or more records to the log. Logging is called "write-ahead" because log records must be safely stored to stable storage before transactions commit, and before the volatile representation of an object is copied to nonvolatile storage. Because of this strategy, there are log records in stable storage for all the changes that have been made to nonvolatile storage, and for all committed transactions. Thus, the log can be used to recover from aborted transactions failures, system failures and media failures. Write-ahead logging is based on either value logging or operation logging. In value logging the old and the new values of an object are stored whereas in operation logging the operations are stored.
Both value and operation logging algorithms require that periodic system checkpoints be taken. In general, checkpointing is an activity that writes information to stable storage during normal operation in order to reduce the amount of log data and shorten the time to recover after a system failure. Checkpointing performs its work by a combination of two types of updates to stable storage [Mullender 89]:

1. marking the log, commit list and abort list to indicate which updates are already written or undone in the stable database.
2. writing the after images of committed updates or before-images of aborted updates in the stable storage.

By using the information written by the checkpoint process we are able to know which transactions to undo and which to redo.

### 2.2.4 Media failures

A media failure is a failure in which some portion of the secondary storage medium is damaged. The recovery process in this case consists essentially of restoring the database from an archive dump and then using the log to redo transactions run since that dump was taken. In general taking a dump will be a fairly lengthy process, much more time-consuming than the operation of taking a checkpoint. Moreover, dumps are normally taken only at very carefully chosen times, such as immediately after a database reorganization, unlike checkpoints which may be taken frequently during normal transaction processing. Other systems support the dumping of just those pages that have been updated since the last dump (incremental dump). Such facilities are essential if the database is very large or if normal transaction processing occupies all available time.

### 2.2.5 The Commit protocol

Failure atomicity requires that database systems provide the all-or-nothing property of transaction executions. In a centralized database this is usually achieved by executing an atomic commit operation. Unfortunately, in a distributed system this atomic operation must be executed over all the sites where the transaction executed any operations. This may cause several complications, especially as some sites may not agree to commit the transaction. This may arise for example due to recoverability considerations. Another significant concern is the distributed nature of failures in a distributed system. Unlike centralized databases where failure is a local event to the sole site of interest, in a distributed database some site may fail while others may remain functional. Several protocols have been proposed
to solve the distributed commitment protocol. The most important is the two-phase commit protocol. It goes as follows [Bernstein 87]:

1. The initiator sends a Vote-Req (i.e. vote request) message to all participants.
2. When a participant receives a Vote-Req, it responds by sending to the initiator a message containing that participant’s vote: Yes or No. If the participant votes No, it decides Abort and stops.
3. The initiator collects the vote messages from all participants. If all of them were Yes and the initiator’s vote is also Yes, then the initiator decides Commit and sends Commit messages to all participants. Otherwise, the initiator decides Abort and sends Abort messages to all participants that voted Yes (those that voted No already decided Abort in step 2). In either case, the initiator then stops.
4. Each participant that voted Yes waits for a Commit or Abort message from the initiator. When it receives the message, it decides accordingly and stops.

The two phases of 2PC are the voting phase (steps 1 and 2) and the decision phase (steps 3 and 4). We see that at various points of the protocol, processes must wait for messages before proceeding. However, such messages may not arrive due to failures. Thus, processes may be waiting for ever. To avoid this, time-outs are used. When a participant does not respond (during the first phase) in a certain amount of time, it is assumed to have failed. The initiator acts as if the participant voted No. The protocol we have studied so far is called the centralized 2PC protocol.

In an attempt to reduce the time and message complexity of the centralized 2PC, two other protocols have been proposed, the decentralized 2PC protocol and the linear (or nested) 2PC protocol [Bernstein 87]. Both have the same fundamental properties as the centralized 2PC, but they use different communications topologies than centralized 2PC. Decentralized 2PC can be used in the star topology. Linear 2PC protocol is usually used in bus topologies. See figure 2.12 for the topologies of these three protocols.

![Figure 2.12](image-url)
Decentralized 2PC works as follows. Depending on its vote, the initiator sends Yes or No to the participants. This message has a dual role: It informs the participants that it is time to vote and also tells them the initiator's vote. If the message is No, each participant simply decides abort and stops. Otherwise, it responds by sending its own vote to all other processes. After receiving all the votes, each process makes a decision: If all were Yes and its own vote was Yes, the process decides Commit; otherwise it decides Abort. Time-out actions can be supplied just as in centralized 2PC.

Linear 2PC is designed to reduce the number of messages. Each process can communicate with its left and right neighbors. The protocol is initiated by the initiator, which is the leftmost process in the linear order. The initiator sends a message to its right neighbor containing its vote, Yes or No. This message informs participant 2 of the initiator's vote and tells it to vote to0. In general, a participant waits for a message from its left neighbor. If the participant receives a Yes and its own vote is Yes, it forwards a Yes to its right neighbor. If the participant receives a Yes and its own vote is No, or it receives a No, then the participant forwards a No to its right neighbor. If these rules are observed, then the rightmost participant will have all the information it needs to make a decision: If it receives a Yes and its own vote is Yes, then the decision is Commit; otherwise the decision is Abort. Having made the decision, the rightmost participant sends a Commit or Abort to its left neighbor informing it of the decision. Each participant that receives the decision message decides accordingly and then forwards that message to its left neighbor. Eventually the message reaches the leftmost participant, at which time the protocol ends.
2.3 Correct executions

In a database system that must correctly handle transaction and system failures (as most do), the executions should be ordered in such a way that we maintain execution and failure atomicity. To provide failure atomicity, executions should be recoverable, avoid cascading aborts or are strict. To provide execution atomicity, executions also should be serializable.

I shall use the notations RC, ACA, ST and SR to denote the set of executions that are recoverable, avoid cascading aborts, are strict and serializable respectively. Recoverability, avoiding cascading aborts and strictness are increasingly restrictive properties:

\[ ST \subset ACA \subset RC \]

The set of serializable executions (SR) intersects all of the sets RC, ACA and ST, but is incomparable to each of them [Papadimitriou86]. Incomparable here means that neither of the RC, ACA, and ST executions are contained in SR and vice-versa. The relationship among the four executions are illustrated in figure 2.13 by a Venn diagram.

![Venn diagram showing relationships between executions that are SR, RC, ACA and ST.](image-url)
According to this Venn diagram we can distinguish between eight (concurrent) executions, as is also shown in the table of figure 2.14.

<table>
<thead>
<tr>
<th>Serializable</th>
<th>non-Serializable</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-RC</td>
<td>non-RC</td>
</tr>
<tr>
<td>RC</td>
<td>RC</td>
</tr>
<tr>
<td>ACA</td>
<td>ACA</td>
</tr>
<tr>
<td>ST</td>
<td>ST</td>
</tr>
</tbody>
</table>

Figure 2.14  The eight possible executions

Assume that we have two transactions T1 and T2 which are as follow:

T1: R1(y)  W1(y)  W1(x)  W1(p)  C1
T2: W2(q)  R2(y)  W2(y)  W2(x)  C2

If we interleave the executions of both transactions we have the following conflicting operations:

R1(y) ↔ W2(y)
W1(x) ↔ W2(x)
W1(y) ↔ R2(y)
W1(y) ↔ W2(y)

In an interleaved execution of both transactions these conflicting pair of operations should be ordered as in a serial execution of T1 and T2 (thus either as in T1, T2 or T2, T1).

We can provide the eight possible interleaved executions of transactions T1 and T2:

Non-RC & SR


This schedule is not recoverable because T2 commits before T1 from which it has read the value of y. In this schedule T1 is serialized before T2.
Non-SR & non-SR


This schedule is not recoverable for the same reason as above.
The edge T1→T2 is in SG because R1(y) < W2(y), and the edge T2→T1 is in SG because W2(x) < W1(x).
Thus SG has a cycle and thus is the schedule not serializable.

RC & SR


This is a recoverable execution because T2 reads from T1 the value of y and it commits after T1. It is however not ACA because T2 reads y from before T1 committed (i.e. uncommitted data). T1 is serialized before T2.

RC & not SR


This is recoverable but not ACA because T2 reads uncommitted data (y). The schedule is not serializable because the schedule contains both the orders R1(y) < W2(y) and W2(x) < W1(x). Thus the SG contains a cycle.

ACA & SR


This execution is ACA but not ST because T2 overwrites the value of y written by T1 before the latter terminates.

ACA & not SR


This execution is not SR because the schedule contains both the orders R1(y) < W2(y) and R2(y) < W2(y), thus the SG contains a cycle.

ST & SR


This execution is both ST and SR. T1 serialized before T2 and T2 reads and writes only from committed data.
ST & not SR


This execution is not SR because the schedule contains both the orders R1(y) < W2(y) and R2(y) < W1(y).
The execution, however, is ST because both transactions read and write only from committed data.
Abstract

This chapter describes the object-oriented analysis and design methodologies. Large and complex systems have enforced in particularly the requirements of reusability, extensibility and encapsulation to provide maintainable and robust software. The traditional software design methodologies could hardly fulfil these requirements. Object-oriented software development methodologies have proven to be better suitable for the analysis and design of large systems. This chapter will handle these methodologies together with the problems of the state-of-the-art object-oriented software development methodologies. As a solution to the identified problems, an extension to the object-oriented model is presented: composition filters.
3.1 Introduction

The object-oriented approach in software development has appeared as an elegant solution for the problems arising from different areas. Object-oriented languages like Smalltalk [Goldberg 83] have emphasized reusability and prototyping as primary software development issues. Systems like Emerald [Black 87], Choices [Campbell 91], [Leyens 89], Apertos [Yokote 92], SOS [Makpangou 88] and Nexus [Tripathi 87] have applied object-based models for constructing distributed systems.

The problems we have to deal with in a software development process, are typically related to large and complex software systems. The characteristics of large software systems impose several requirements on the system development: namely [Johnson 88]:

- **Encapsulation.** The work must be divided among different people. One person should not have to know the details of units built or modified by other people, but just how the unit he is responsible for interacts with the rest of the system.

- **Reusability.** Developing a large system is expensive. The development cost can be reduced if some of the units that the system is built from could be taken from already existing systems. Similarly, it would be beneficial if some parts could be reused in future projects.

- **Extensibility.** Due to the high development cost, large software systems are usually long-lived. During its lifetime, the system undergoes considerable modification because of the changing specifications. Therefore, the system should be easily modifiable and extensible.

The advantages of object-oriented software development is their support for encapsulation, reusability and extensibility [Micallef 88]. These issues are interrelated with the requirements for large software systems.

**Encapsulation**, is the strict enforcement of information-hiding. Encapsulation hides unnecessary detail and since it explicitly divides the interface and implementation parts of units, the reimplementation of these units will have no effect on the other objects in the system.

**Reusability** is the ability of a system to be reused, in whole or in parts, for the construction of new systems.

**Extensibility** is the ease with which a software system may be changed to account for modifications of its requirements.

The requirement of maintainability is closely related with the indicated requirements. Software main-
Chapter 3 Object-oriented Analysis and Design

tenance can be categorized as corrective, adaptive and perfective [Jonson 88]. Corrective maintenance is the process of diagnosing and correcting errors. Adaptive maintenance consists of those activities that are needed to properly integrate a software product with new hardware, peripherals, etc. Perfective maintenance is required when a software product is successful and its functionality ought to be extended. Ensuring the requirements for encapsulation, reusability and extensibility will provide software which is robust and better maintainable.

The rest of this chapter is organized as follows: In the following section we introduce the basic-concepts of the object-oriented methodology: abstraction, objects, classes, inheritance and polymorphism. This will be followed by section 3.3 which will give a description of object-oriented software development system followed by section 3.4 with the problems of current object-oriented methodologies. The chapter will be concluded with section 3.5 which handles the composition-filters approach.

3.2 Object-oriented concepts

Below the main concepts in object-oriented software development will be described. These are objects, messages, classes, abstraction, inheritance, delegation and polymorphism.

Object
An object is a well-defined data structure coupled with a set of operations that describe specifically how that data can be manipulated. The behavior of an object is characterized by the operations defined on it; this means that only these operations can manipulate the object. A message is a request for an object to carry out one of its operations. A message specifies which operation is desired, but not how that operation should be carried out. The receiver object has methods which describes how the operations are performed. The set of objects to which an object can respond is called its interface with the rest of the system. The only way to interact with an object is through its interface.

Class
A system will often contain many similar objects. A class describes the implementation of a set of objects that all represent the same kind of system component. The individual objects described by a class are called its instances. A class describes the form of its instances’ private memory, called instance variables, and methods that manipulate the instances. Only the methods of the class can access the instance variables directly; other messages must use message sending to gain access to or update the value of the instance variables. All instances of a class respond to the same messages; they can only differ in the value of their instance variables. A class that defines an interface but only a partial implementation is called an abstract class. An
abstract class can not have an instance since its implementation is not complete. A class that does provide a complete implementation is called concrete and may have an instance. Generally it is good practice to represent the design by an abstract class and provide the implementations as concrete subclasses. This allows for the separation of the interface needed for proper use of a class and the implementation details in the definition of a new subclass. Figure 3.1 illustrates the basic principles of the object-based language model.

**Abstraction**

Using abstraction is one of the fundamental ways we can cope with complexity. Abstraction denotes the essential characteristics of an object and hides the unnecessary detail. An abstraction focuses on the outside view of an object and so serves to separate an object’s essential behavior from its implementation. The objective of abstraction is often to consider a class of objects and to consider only those characteristics of the objects in the class which are common to all of them, thereby abstracting from characteristics in which objects differ.

**inheritance**

Simply stated, inheritance is a relationship among classes wherein one class takes over (inherits) the structure or behavior of another class. Each class has a superclass from which it inherits operations and internal structure. A class that inherits from a superclass is called a subclass. The subclass can add new instance variables and methods of its own. It can also define a method with the same selector as one of
the superclass’s methods; this is called overriding. The subclass on his turn may pass on its inherited characteristics to its subclasses. Inheritance allows common behavior from related classes to be factored out and implemented only once in a common place, the superclass. This reduces the size of the subclasses thereby decreasing code maintenance efforts. Inheritance from more than one superclass is called multiple inheritance.

**delegation**

Delegation is a mechanism that allows objects to delegate the request of its users to one or more designated objects [Lieberman86]. Hereby the designated objects are part of the extended identity of the delegated object. Delegation is defined independently of the class concept and often used by classless languages.

Delegation is defined to be a form of resource sharing that allows shared resources to be viewed as belonging to the object on behalf of which they are executed. Thus, a given operation can “belong” to different objects on different instances of execution.

Delegation can support dynamic evolution of systems because delegations can be configured at runtime. Moreover, delegation can be used for both behavior as state sharing, whereas inheritance only supports sharing of behavior.

**Polymorphism**

In general “polymorphism” means the ability to take over several forms. In object-oriented software development, this means that the same message may behave differently on different classes. The most simple view of polymorphism is sending of a message to different objects that are not related, but can still respond to the same message. Although the message is the same, the methods that will be executed depend on the receiver object. A more interesting form of polymorphism is polymorphism through inheritance, whereby a message is redefined through the application of the inheritance mechanism. We call an operation in a class polymorphic if it can be applied to different subclasses. In addition to polymorphic operations we have polymorphic variables like `self` which refers to the receiver of the message that invoked that method.
3.3 Object oriented software development

The object-oriented software development process can be sub-divided into analysis, design and implementation phases.

The analysis phase is concerned with the precise and correct identification of the user-requirement’s in an understandable way. It aims to express only the properties of the real system insofar as they are relevant to the future users. The analysis phase describes what properties the real system must possess, not how these properties are achieved. This implies that the analysis phase must result in a model that fulfils the user requirements but also refrains from expressing any detail that may only be relevant in consecutive phases.

The design phase reflects a structure of descriptive elements and combinations thereof that can be directly and efficiently mapped onto physical and logical elements that will embody the real system. The design phase may revise and extend the analysis model to specify how the user requirements can be realized. The revision and extending of the analysis model may be based on criteria like correctness, robustness, extensibility and reusability.

The implementation phase provides the mapping from the design model onto physical and logical elements embodying the real system.

To manage with the complexity different software methodologies have been developed. The original waterfall model views the software process as being made up of a strictly ordered number of stages [Sommerville 82]. After each stage is defined it is “signed off” and development proceeds to the following stage. This process is illustrated in figure 3.2.

However, this development model is only appropriate for some classes of software systems. It is useful for educational and management purposes to consider the phases of the software life-cycle to be distinct but, in practice, the development stages overlap and feed information to each other. While a design is being developed, problems with requirements are identified; while a program is being coded, design problems are found, and so on.

To solve these problems software engineers use the waterfall with iteration model [Sommerville 82]. In this model the software engineer is allowed to iterate between different phases. This is illustrated in figure 3.3. Because the waterfall with iteration model is the most representative for flexible software development most object-oriented methodologies use this model.
In traditional development methodologies, the conceptual models used for analysis differ from those used for design. Programming has yet a third view of the world. Analysts use entity-relationship models, functional decomposition and matrices. Designers use data-flow diagrams, structure charts and other diagrams. Programmers use the constructs of different languages like Pascal, C, Cobol, Fortran etc. In object-oriented techniques analysts, designers, implementers and particularly important the end-users all use the same conceptual model. They all think in terms of the basic building bricks of objects and classes. As all the parts involved in the software development system use similar models...
3.4 Problems in the state-of-the-art methods

Each phase in object-oriented software development can be divided into three subcomponents: preparatory work, structural relations and object-interactions [Aksit 92a].

**preparatory work**

The preparatory work in the analysis phase is concerned with the mapping of real world entities to the objects in the analysis model. This mapping process is called domain analysis. In addition, the problem domain will be partitioned into manageable sub-components called subsystems. Since the object-oriented constructs simulate corresponding entities in the real world the problem domain knowledge can directly be mapped to an object-oriented analysis model.

The preparatory work for the design phase consists mainly of mapping the objects in the analysis model to predefined classes in the design environment. Another activity is collecting and formulating design requirements needed for efficiency and alternative realizations.

The preparatory work for the implementation phase is concerned with the environment within which the design will be implemented.

**structural relations**

The two most important relations used in the analysis phase are classification and part-of relations. In addition we can identify associations which are used by some methods to describe relations other than classification and part-of relations. Classification relations reflect generalization and specialization relations between different classes. Part-of relations indicate part-whole relations between the classes. Subsystem-partitioning in the analysis phase often matches part-of relations.

In the design and implementation phases these relations are manifested by means of class-inheritance hierarchies.

**object interactions**

The structural relations define the architecture of a system. The dynamic behavior of the system is realized by object interactions. These are represented by message connections which simply indicate the communication between two objects.

In the analysis phase object-interactions are represented by functions that an object should provide and additionally from which these requests can be expected. In the design and analysis phases, object interactions may be modified as a result of various design decisions like improving reusability and
Although many object-oriented software development methodologies claim to support the creation of robust and maintainable software there are several obstacles which they don’t address. The paper [Aksit 92a] describes the obstacles in the object-oriented software development process. These obstacles will be handled in the next sections.

### 3.4.1 Problems related to the preparatory work

When we try to fully understand a large and complex problem domain we can identify six obstacles. The first two problems are related with the domain analysis. The remaining four problems are related to subsystems.

1. **Identification of problem-domain structures**
   During the analysis phase we try to identify classification in the problem domain to map them directly into inheritance hierarchies. This is manageable for some well-structured and theoretically founded domains. However, very often underlying theories of large systems are not completely understood, and it is difficult to define reusable hierarchies for these types of systems. The lack of insufficient and unclear domain knowledge are the main reasons for bad structuring of these systems.

2. **Dealing with excessive domain objects**
   This problem denotes not the lack but the excessive domain knowledge. If the software engineer uses its complete domain knowledge of a problem to specify the user’s requirements, this may result in an excessive number of objects, although only a few of these objects may be relevant for the problem being analysed.

3. **Early decomposition**
   To deal with a large number of objects in a system the software engineer can partition the application into subsystems prior to the object identification phase, and only then consider the objects within the context of these subsystems. However, this may result in subsystems which boundaries may not be optimal since subsystem boundaries are mainly determined by inter-object relations and interactions. Thus, the problem here is that if we start object-identification before defining subsystems the project may become unmanageable. On the other hand, if we identify subsystems prior to object identification then the defined subsystem boundaries may not be optimal.
4. **Subsystem-Object distinction**

Most object-oriented software development methods consider subsystems as being different from objects. In some methods, subsystems are used to collect related objects. The problem here is the distinction between subsystems and objects. During the analysis phase, objects may eventually act as subsystems if their internal structures get too complicated. Similarly, subsystems may act as objects if their functionality can be structured in a class hierarchy and reused in different applications. Since object-oriented methods are largely iterative, one may need to convert subsystems to objects or vice versa. This requires modifications to the semantics of these constructs, which is obviously very error prone.

5. **Commonality versus Partitioning**

In large and complex systems, subsystems are assigned to different software engineers, and/or handled sequentially one at a time. However, this will make the process of identification of class hierarchies very difficult since classes belonging to the same hierarchy can be scattered over different subsystems.

6. **Subsystems identification using object interactions**

The identification of subsystems is mainly used for structuring interactions among objects. However, in most existing methods, this is largely based on intuitive techniques. This is not sufficient for large systems, since interactions can be too complex and subject to changes.

3.4.2 **Problems related to the structural relations**

1. **Sharing behavior with state**

For certain applications, it may be desirable that the state shared by instance objects affects their operations defined at the class level. The required concept here is delegation which is not available in some object-oriented methods.

2. **Atomicity versus inheritance**

Most systems provide transactions for a program block by delimiting it with `StartTransaction` and `EndTransaction`. Although transactions are useful abstractions to preserve consistency, they are not uniformly integrated with the object-oriented concepts. For example, a class would have to declare atomic transaction blocks for all the combinations of inherited operations from its superclass.

3. **Arbitrary inheritance mechanisms**

We can use class inheritance to exclude, override and/or extend the operations and local variables of
the superclasses. However, we may also need inheritance hierarchies which require semantics other than overriding or extending operations. This problem can be exemplified by the inability to redefine a grammar specification using an inheritance mechanism.

4. Inheritance versus states
The notion of states in object-oriented methods is used for capturing the dynamic behavior of systems and to identify the operations of objects. In general, a state represents the condition of an object expressed in terms of the values of local variables. The problem here is the lack of integration between the notion of states and the inheritance mechanism. If state specifications are not allowed to be inherited this will result in bad reusability. On the other hand, if state specifications are inherited by subclasses, it may cause problems in the extensions made to the original specifications.

3.4.3 Problems related to object interactions

1. Multiple views
Since not all the operations of an object are necessary for all objects it serves, it is desirable to define views on an object, differentiating between clients. However, most object-oriented methods cannot express multiple views.

2. Queries and Language-Database integration
It is preferable to integrate database issues like persistent data structures, transactions and queries in software development [Aksit 91]. Object-oriented methods does not support this yet. The problems that cause the inability to complete integration of language and database systems are many-fold. Firstly, we are dealing with two different systems, since these systems extend an object-oriented computation model with conventional database mechanisms. Secondly, introducing database-like features into the object-oriented language model generally weakens encapsulation. Thirdly, in most systems queries are restricted to a fixed number of classes and thus objects to be accessed have to be inserted into an instance of these classes explicitly.

3. Coordinated behavior
The object interaction in object oriented methods is mainly based on message invocations between two objects. This message send model can only specify communications that involve two partner objects at a time and its semantics cannot be easily extended. Mechanisms like inheritance and delegation only support the construction and behavior of objects but not the abstraction of communication among objects. These mechanisms therefore fail in abstracting patterns of mechanisms and larger scale synchronization involving more than just a pair of objects.
3.5 The Composition-filters model

Although the traditional object-oriented model has obvious benefits like providing encapsulation, data abstraction and inheritance, it has appeared that it cannot cope with many of the problems denoted in the last section. The Trese-group at the University of Twente has developed the concept of composition-filters as an extension to the traditional model to encompass these problems [Aksit 91], [Aksit 92b], [Bergmans 92]. The composition filters model is adopted by the object-oriented language Sina. This section will handle this model.

3.5.1 The basic object model

As illustrated in figure 3.4, a composition-filters object is subdivided in two parts: an interface and an implementation part.

Figure 3.4  The components of the composition filters object model
The interface part contains the declarations of the interface objects, the class-specific methods, conditions and the input- and output filters. The interface object can be divided into two components. The first component consists of encapsulated interface objects called internals. The second component consists of interface objects that are outside, but within the scope of the object. These are called externals. The incoming messages are handled by input- respectively output filters. The conditions together with the filters decide whether the requested message will be executed or not.

The implementation part contains method definitions, instance variable declarations, definitions of conditions and an optional initialization method. The implementation part is fully encapsulated within the object.

3.5.2 The Interface part

As an example of a simple class, consider the interface part of class TransactionManager in figure 3.5. The examples will be presented following the Sina language.

```sina
class TransactionManager interface
  comments    This class implements a TransactionManager
  conditions
    busy;   // This condition is only valid if startTransaction has been invoked before
  methods
    startTransaction returns Nil;
    commit returns Nil;
    abort returns Nil;
  inputfilters
    disp: Dispatch = | busy => * \ inner.startTransaction;
                   True => inner.startTransaction
end;
```

Figure 3.5 A definition of the TransactionManager class

The methods that are to be visible at the interface of the object are declared in the interface part by method headers following the keyword methods. Class TransactionManager for instance, declares the methods startTransaction, commit and abort. As we can see, method declarations in the interface part only give names, argument types and return types of methods that are available to users of the object. The actual implementations of the methods are encapsulated within the implementation part. To invoke one of these methods, an appropriate message must be sent to an instance of class TransactionManager.
When an object receives a message from outside the object boundaries it directly submits this message to its input filters. If the message is invoked within the object’s boundaries and is sent to an object outside of the object boundaries, the message will be submitted to the output filters. The invoked message will only result in the execution of a method, after the message successfully passes all the input filters in the interface part of the object.

After the keyword input filters, class TransactionManager defines one input filter, called disp, of class Dispatch using the expression:

```
disp: Dispatch = ....
```

An input filter of class Dispatch is used to initiate execution of a method if the message successfully passes it. After the “=” character the filter elements of the defined filter is specified:

```
disp: Dispatch = { busy => * \ inner.startTransaction,
                   True => inner.startTransaction }
```

The filter elements are separated by commas. Each filter element consists of a condition and a matching part. The condition is defined at the left hand side of the “=>” character. The matching part is placed after the “=>” character.

Actually, conditions are like logical propositions. Their value is either true or false. The names of the conditions are declared in the interface part following the keyword conditions, and their definition is provided in the implementation part. Conditions may reflect the values of instance variables, but may also reflect external variables. In this example, we have one filter called disp which is an instance of the Dispatch filter class. This filter contains two filter elements, with the conditions True respectively busy. The condition busy is set to true if the message startTransaction is invoked. The condition True preceding each filter element means that the matching part of the filter element will always be checked.

The matching part of a filter element is composed of a target with a selector. The received message is first matched with the selector specified by the filter element. If this match is successful, then the target specified by the filter element is bound to the message. The character “*\inner.startTransaction” in our example indicates that all messages are acceptable excluding message startTransaction. The pseudo variable inner denotes the methods defined by TransactionManager.

A received message is evaluated by all the input filters of an object. The evaluation is done for each filter in left-to-right order of the filter elements. A message is accepted by an input filter if it matches
with the matching part of a filter element for which the condition value is true.

A filter can be treated as an ordinary language object that determines which action must be taken when a message is accepted or rejected, respectively. Each filter is declared as an instance of a filter class. A programmer is allowed to define an arbitrary number of filters for an object. Each filter can be an instance of an arbitrary filter class. The complete set of input filters of an object determines the conditions for message acceptance, and determines which method will be executed upon acceptance.

A general form of a filter initialization is as follows:

```
aFilterName: aFilterClass = {aCondition1 => [aMatchTarget1.aMatchSelector1] aSubstTarget1.aSubsSelector,
...... |;
```

Here, the filter `aFilterName` is declared as an instance of class `aFilterClass`. A filter is initialized by a sequence of filter elements separated by commas. A filter element defines a specific condition for accepting a particular set of messages. Each filter element consists of a condition part `aCond1` and a message part `[aMatchTarget1.aMatchSelector1] aSubstTarget1.aSubstSelector1", as shown above. The matching part between the brackets is optional.

The basic mechanism as presented here has some extra features, most of them for convenience:

- If no condition is specified then this will be treated as if the condition is True.
- The character "*" may be used to denote a don’t care condition
- The character "\" may be used to exclude messages from the matching part of a filter element
- If the target part is omitted, the pseudo-variable self is assumed.
- To shorten filter expressions, one can combine several filter elements together.

For example, instead of a single condition, a set of conditions can be provided, as follows:

```
{ condition1, condition2 } => aTarget1.aSelector1
```

is equivalent to

```
condition1=>aTarget1.aSelector1, condition2=>aTarget1.aSelector1
```

- It is also allowed to use a short-hand notation when a single condition corresponds to several target-selector pairs:

```
{ condition1=>[aTarget1.aSelector1, aTarget2.aSelector2] }
```

is equivalent to

```
{condition1=>aTarget1.aSelector1, condition1=>aTarget2.aSelector2 }
```
3.5.3 The implementation part

The components of the implementation part are exemplified by class TransactionManager in figure 3.6. Instance variables are declared in the insvars clause. Instance variables are fully encapsulated and can be objects of arbitrary complexity. Class TransactionManager declares 2 instance variables named timestamp and busytrue. These instance variables may be accessed directly only by the methods defined within the object’s class, external clients of an object, or even its subclasses cannot do this.

The conditions are declared in the conditions clause. The implementation of the conditions are defined message expressions. A condition implementation is similar to a method. However, a condition implementation always results in a Boolean value and is free of side effects.

The initialization method of an object is defined in the initial clause. This method is executed immediately after object creation.

The last component of the implementation part is the definition of the methods. A method consists of a series of message expressions. The control flow may be controlled by a set of standard control statements.

```
class TransactionManager implementation
   comment this is the implementation of TransactionManager
   instvars
      timestamp: Integer;
      busyTrue: Boolean;
   conditions
      busy
         begin  return busyTrue end;
   initial
         begin busyTrue:=false end;
   methods
      startTransaction
         begin ...; busyTrue:=true; ... end;

            commit
               begin ...; busyTrue:=false; ... end;
            abort
               begin ...; busyTrue:=false; .... end;
   end;
```

Figure 3.6 The implementation part of class TransactionManager
3.5.4 Inheritance and delegation

In the composition-filters model, inheritance is not directly expressed by a language construct, but is simulated by input filters. The class from which must be inherited should be declared as an internal object. By delegating messages to the methods provided by this internal object, inheritance is simulated. If the ‘superclass’ is declared as an external then this will provide delegation. We extend the TransactionManager class to illustrate the concepts of inheritance and delegation (see figure 3.7).

```plaintext
class TransactionManager interface
    comments    This class implements a TransactionManager
    internals
        senderCommit: SenderCommitProtocol; // instance of the ‘superclass’
    external
        policyManager: PolicyManager; // instance of the delegated server
    conditions
        busy; // This condition is only valid if startTransaction has been invoked before
    methods
        startTransaction returns Nil;
        commit returns Nil;
        abort returns Nil;
    inputfilters
        disp: Dispatch = [True => senderCommit.*, 
                            True => policyManager.*, 
                            busy => * \ inner.startTransaction,
                            True => inner.startTransaction]
end;
```

**Figure 3.7** class TransactionManager illustrating delegation and inheritance

In figure 3.7 we have declared an internal object of class SenderCommitProtocol and an external object of class PolicyManager. The filter disp of class Dispatch contains three filter elements. The third and fourth filter elements were described before.

The first element of the filter specifies that all incoming messages are delegated to the internal object senderCommit, provided that these messages are supported by class SenderCommitProtocol. Since the methods of SenderCommitProtocol are now available to TransactionManager through an instance of SenderCommitProtocol, class TransactionManager inherits the operations of class SenderCommitProtocol. This inheritance mechanism is also referred to as delegation-based inheritance.

The second filter element of the filter specifies that all incoming messages are delegated to the external object policyManager. The mechanism here is the same as above. The only difference is that the mes-
sages are delegated to an external object in place of an internal object. This provides us delegation.

When an instance of class TransactionManager is created, its internal object senderCommitProtocol is also created. However, external declarations do not result in automatic object creation. In addition, because external objects are not encapsulated within the object, it can be shared by other objects. The class PolicyManager here serves different TransactionManager objects and stores shared data for these objects (like a local timestamp counter).

Multiple inheritance can be provided by declaring multiple internal objects. It is also possible to have multiple external objects to delegate the messages to. The left-to-right evaluation order avoids name conflicts, if any.

3.5.5 Concurrency through input filters

In Sina there are two ways to achieve concurrent threads. Whenever an object is created, its (optional) initial process will be activated as an independent process. This process may remain active for an arbitrary length of time.

A second possibility to achieve concurrency is by using early return statements. In early returns the sender object sends a message synchronously and is blocked from carrying out further operations until it has received a reply from the receiver of the message. The receiver object can execute a reply statement somewhere during its execution. In this case, a result is returned to the sender object and both objects continue with their executions concurrently.

In Sina there can be more active threads within an object. Consistency can be guaranteed by defining the appropriate filters. By default, the pre-processor of the Sina language includes for every object a filter which provides mutual exclusion, allowing only one active thread within the object. This can be undone by specifying a compiler option.

Object Manager

Every object has an object manager, which is a system-defined object that is responsible for the scheduling and processing of the requested messages to the object it manages. The messages are actually first received by the object manager. The object manager then places the received message queue and removes it from the queue when it can be activated. In addition, the object manager initiates filters and keeps information about the active threads.
The object manager also keeps information about the number of active and blocked message request. This information can be obtained by sending messages to it. An object can send messages to its object manager by specifying the identifier "^self" or "^server" as the target of a message invocation. An object manager is encapsulated within its object and can only be accessed by its own object. For example, we can request the number of active threads in the object by means of the following expression:

```
^self.active
```

Mutual exclusion can be defined as follows:

```
conditions
  Free;       // indicates that there is currently no active thread within the object
inputfilters
  mutex:Wait = {Free => * };
```

Whereby the condition Free is implemented as follow:

```
Free
  begin  return ^self.active=0 end;
```

If a message is submitted to the filter `mutex`, first it will be checked whether there is an active thread within the object. If this is the case then the message will be blocked until the object is free.

### 3.5.6 Synchronization through input filters

The synchronization between active processes within an object can be specified by filters of class `Wait`. When a message is accepted by a filter of class `Wait`, the message passes to the next filter. If, however, this filter does not accept the received message, the message will remain in the queue until it will match with a filter element. Requests to methods of an object can be synchronized by associating messages with specific conditions that implement specific synchronization conditions.

We shall explain the `Wait` filter by means of the class `LockManager` in figure 3.8. This class provides two methods, lock and unlock, which respectively lock and unlock the object. The method unlock is always available, independent of the state of the object. The other methods of the objects are only allowed when the object is in the unlocked state. If a lock message is received while the object is in a locked state then the `Wait` filter will cause this message wait in the message queue until the object is unlocked again.
class LockManager interface
  methods
  lock returns nil;
  unlock returns nil;
  conditions
  Unlocked; // true if object is locked
  inputfilters
  sync: Wait = { True => unlock, Unlocked => * }
end;

class LockManager implementation
  instvars
  free: Boolean;
  conditions
  Unlocked begin return free;
  methods
  lock begin free:=false end;
  unlock begin free:=true end;
end;

Figure 3.8 The definition of class LockManager

3.5.7 Abstract Communication Types

An ACT class is an ordinary Sina class which uses Meta filters to abstract the communications and to coordinate the behavior among objects. The Meta filter is used to convert a message into its first-class representation. If the received message is accepted by the Meta filter it first converts it to an instance of class Message and then passes it to the ACT object which is responsible for the coordinated behavior. The ACT object on his turn can perform operations on the message like modifying the contents of the message by invoking the operations of class Message. In addition the ACT object can convert an instance of class Message back to a message execution. Thus, actually ACT classes accept the messages submitted by Meta filters. ACTs can be declared as internal or external objects, depending on the kind of applications.

Consider the following example:

    aMetafilter : Meta = [ aCondition => [self.aMethod] anACT.aMethodOfanACT ];

This expression is similar to the definition of an ordinary filter, except the matching part is now between the brackets “[“ and “]” and an extending part has been added. If the message is accepted in the filter above, that is, if both aCondition is true and the message is self.aMethod, then the message will be converted to its first-class representation. The new instance of class Message will be submitted to
the ACT class object \textit{anACT} by invoking the message \textit{anACT.aMethodOfAnACT(aMessage)} where the argument \textit{aMessage} is the class-representation of the original message. If the received message does not match with a \textit{Meta} filter it is passed to the next filter. We can illustrate the ACT concept with the class 

\begin{verbatim}
class ApplicationWithTransaction interface
  internals
    tm: TransactionManager;
  methods
    transmethod returns Nil;
    commit  returns Nil;
    abort   returns Nil;
  inputfilters
    disp: Dispatch = { True => *,*} 
  outputfilters
end;
\end{verbatim}

\textbf{Figure 3.9} The interface of the class ApplicationWithTransaction

The method transmethod contains the actual implementation of a transaction. The user initiates its transaction by invoking the message transmethod on the application object of class ApplicationWithTransaction. This will pass the inputfilters without any problems. In the transmethod subsequent messages will be invoked. If these invoked messages are outside the object boundaries then they will be captured by the outputfilter \textit{constr}. The outputfilter will convert each of these messages to its class representation and then will pass it as an argument of the handleMessage to the internal ACT object \textit{tm}. The ACT object will perform operations that are needed for each transaction message. See figure 3.10 for the graphical representation for this construction.

\begin{verbatim}
messages
    Dispatch-filter
   /    \rightarrow
  tm \\
/    \leftarrow
Meta-filter
\end{verbatim}

\textbf{Figure 3.10} Outgoing messages are delegated to the internal \textit{tm} ACT object
3.5.8 Multiple Views through filters

In Sina it is possible to provide multiple views for an object. The differentiating of the interfaces of the server object may depend on its state or on characteristics of its clients. The server object may for example make some methods visible to instances of one class, and others to instances of another class. To provide multiple views the object needs to declare a filter of the class Error. This filter class is used for the selection and rejection of messages only. When a message is rejected by the filter it raises an error condition. We can illustrate the multiple view mechanism with the class TransactionManager in figure 3.11.

In this example two views on the class TransactionManager are defined. Objects of class ApplicationWithTransaction are allowed to invoke all the messages except setTimestamp while objects of class PolicyManager or its subclasses are only allowed to send the message setTimestamp. The pseudo variable sender is used to check the class of the client object in the implementation of the conditions application_view and system_view. Note that only an object that is an instance of class ApplicationWithTransaction and PolicyManager or their subclasses may invoke messages on an object of class TransactionManager.

class TransactionManager interface
    conditions
        application_view;
        system_view;
    methods
        startTransaction returns Nil;
        commit returns Nil;
        abort returns Nil;
        setTimestamp returns Nil;
    ......
    inputfilters
        er:Error = { application_view => */setTimestamp,
                    system_view => setTimestamp}
    end;

class TransactionManager implementation
    ....
    conditions
        application_view
            return sender.subtypeOf(ApplicationWithTransaction);
        system_view
            return sender.subtypeOf(PolicyManager)
    ......
end;

Figure 3.11  Class TransactionManager illustrating multiple views
3.5.9 Associative inheritance through filters

With associative inheritance or associative delegation we mean the ability to dynamically configure the behavior of an object by restricting the set of interface objects. We speak of associative inheritance if for associativity internal objects are used. Associative delegation is meant if we use external objects to provide associativity. The client or the object itself may affect the inheritance (or delegation) web to some extent, and specify associatively the objects from which it would like the server object to inherit (or delegate). This can be done by appending the following expression to the inputfilters clause.

```
{ aCondition(#) => #.* }
```

In addition the server object should provide a message to modify the proposition `aCondition`. We illustrate this in figure 3.12 with the class `ApplicationWithTransaction`, which provides a different behavior, depending on the kind of TransactionManager the user wishes his transaction to be handled by. The latter proposition can be determined by the client by sending the message `select_TransactionManager`, providing the proposition as a 'block' argument. Note that in class `ApplicationWithTransaction` the criterion for associativity is only determined by the server object.

```plaintext
class ApplicationWithTransaction interface
  internals
twoPLTM: TwoPLTransactionManager;
toTM: TOTransactionManager;
optimisticTM: OptimisticTransactionManager;
methods
  select_TransactionManager(Block) returns Nil;
conditions
tm_state;
inputfilters
  fl:Error = { True => inner.*, tm_state(#) => #.* };
end;
class ApplicationWithTransaction implementation
  ....
  methods
    select_TransactionManager(new_prop:Block)
    begin
      tm_state.proposition(new_prop);  
    end;
end;

Figure 3.12 Definition of class `ApplicationWithTransaction` which associatively inherits from various' types of TransactionManager object
```
The class ApplicationWithTransaction declares three interface objects: twoPLTM, toTM and optimisticTM of classes TwoPLTransactionManager, TOTransactionManager and OptimisticTransactionManager, respectively. All these classes are specific TransactionManagers which have different semantics.

The user can now specify its required TransactionManager for his initiated transaction by invoking the message select_TransactionManager on an object of the class ApplicationWithTransaction. The new proposition for the behavior condition tm_state can be provided as an argument of the select_TransactionManager method. The method proposition sets the new proposition for tm_state. An example of invoking select_TransactionManager, using an instance of ApplicationWithTransaction called anApplicationWithTransaction, is:

```
anApplicationWithTransaction.select_TransactionManager ( [#.subtypeOf(TwoPLTransactionManager) ] );
```

As we can see the argument is specified as a Block, which is denoted by “[...]”. The number symbol “#” refers to the argument of the proposition (interface objects will be substituted here). By invoking this message the user has enforced the object only to inherit from classes that are subclasses of the class TwoPLTransactionManager. For more details about Associative inheritance I refer to [Aksit91].
Chapter 4

Design of the Framework

Abstract

This chapter describes the design of an object-oriented framework for atomic transactions. The framework can be used in a centralized as well as in a distributed environment. The software development is not based on any rigorous design methodology. The resulting framework provides a flexible, object-oriented interface for creating atomic transactions.


4.1 Introduction

We can identify the following obstacles in traditional transaction systems:

1. Lack of modularity
   The whole transaction system is organized as one big program consisting of many lines of code. We are not able to easily identify the different transaction semantics in the transaction system. This provides bad extensibility when the specification changes. Since extensibility is one of the requirements for maintainability this will result in a system that is also hard to maintain.

2. Lack of diversity
   In all the existing transaction systems a single transaction semantic is adopted. However, different applications may require different transaction semantics. For example an application may require that its operations are handled by a weaker concurrency control method than the transaction system supports (e.g. locking techniques). In order to solve this problem a new transaction system with different transaction semantics must be rewritten while we are not able to reuse the semantics of the existing system.

3. Lack of flexibility
   The conventional transaction systems are not flexible enough to vary the concurrency control and recovery methods. This may be needed for example when during the processing of transactions the performance goes down because of the bad functioning of the adopted transaction semantics. However, the existing transaction systems cannot switch to other transaction semantics even when this is strongly required. The user, the environment and the data may require other transaction semantics than the transaction system is using.

4.2 Requirements

In order to solve the problems above we have the following requirements for a transaction system:

1. Decomposition of the transaction system in manageable components.
   The whole transaction system must be decomposed in manageable components which perform the different transaction semantics. Each of these components should strongly encapsulate their own transaction semantics.
2. **Different transaction semantics provided in a framework**

The transaction semantics should be provided by a framework that consists of classes and class hierarchies. The transaction semantics must be specified by the interfaces of abstract classes while different protocols are implemented in derived subclasses. This makes it easy to select from many different specialized objects because they all conform to a single interface.

3. **Dynamic transaction semantics tuning at three levels**

The transaction framework should permit the user, the environment and the accessed data to participate in the choice of transaction semantics. Based on the dynamically changing conditions of the user, the environment and data the transaction mechanism must be able to switch to other transaction semantics when required.

### 4.3 Decomposition of the transaction system

Figure 4.1 represents the decomposition process in different transaction components.

![Figure 4.1](image)

**Figure 4.1** The decomposition of the transaction system in different components
First of all we have to identify which components we will need.

**Application object**

A user should be able to define its transaction somewhere. Therefore, the user should provide an application object that performs its transaction (see figure 4.2). The application object in our design provides the operation `transaction` where the actual transaction of the user is implemented. So the user has a uses relation with the application object.

![Figure 4.2 Uses relation between user and application object](image)

**TransactionManager**

The user starts its transaction by invoking the message `transaction` of the Application object. The user must be able to invoke data and transaction operations. Transaction operations are: `startTransaction`, `commit` and `abort`. These operations are performed by an object called TransactionManager.

The application object can either have a uses relation or an inheritance relation with the TransactionManager. In our case we must have an inheritance relation. There are different reasons for this decision. If the application class has a uses relation with the TransactionManager then we have hard-coded implementation of the transaction operation in the application class. If in the future versions we may wish to adopt a different TransactionManager but still reuse the same application then we must change the implementation of our transaction. However, if we use an inheritance relation rather than a uses relation between the application object and TransactionManager object we are able to use the polymorphic message passing mechanism which enables us to replace the existing TransactionManager by another TransactionManager provided that they have the same interface.

In the composition filters model we can model the relation between the application and TransactionManager object as illustrated in figure 4.3:
Chapter 4  Design of the framework

By the handling of the transaction method we need also a mechanism to handle the data operations. In our design we used the properties of Abstract Communication Types. Each invoked data operation is captured by the TransactionManager. Therefore, it is necessary that the application has an inheritance relation rather than a uses relation with the TransactionManager. The TransactionManager must provide a method `handleMessage` to handle the operations received from the applications.

The framework should provide a class hierarchy for TransactionManagers using different semantics. The user must then choose one of the existing TransactionManagers. If the user is not satisfied by the existing TransactionManagers then he should be able to subclass from the TransactionManager class hierarchy to provide a suitable TransactionManager. This is also illustrated in figure 4.3.

PolicyManager

Each transaction should have a unique timestamp in our system, because in most concurrency control methods timestamps are used to define a consistent order. This leads us to the identification of another component in our transaction system. We call this component the PolicyManager. The PolicyManager may also adopt different transaction semantics. Therefore we should have a class hierarchy for the PolicyManager too.

Since the PolicyManager will be needed to share information for the TransactionManagers a TransactionManager should have a delegation relation with the PolicyManager. This is shown in figure 4.4. Note that we may create a different transaction system if we adopt a different PolicyManager.

---

**Figure 4.3** Relation between the application object and TransactionManager object
In our transaction system we have also data objects that are accessed. Each data object must be accessed in a consistent order. We have seen that consistency can be violated by concurrency and failures. Therefore, each data object must encapsulate components that contain the transaction semantics for concurrency control and recovery techniques. We call the component that is responsible for concurrency control the Scheduler. The component that is responsible for recovery is called the RecoveryManager.

Because the Scheduler and the RecoveryManager are related to and access the same data object their actions have to be synchronized in some way. The component that provides this functionality is called the DataManager. The DataManager also has the task to capture incoming transaction messages and to choose an adequate scheduler.

Figure 4.5 illustrates the relations between a data object, DataManager, Scheduler and RecoveryManager in the composition filters model.
Chapter 4  Design of the framework

Figure 4.5  The relations between DataManagerObject, DataManager, Scheduler and RecoveryManager

Note that in figure 4.5 a data object is called a DataManagerObject. The incoming messages are send to the DataManager which is provided as an internal of the data object. The DataManager on his turn uses the internals Scheduler and RecoveryManager. Our framework should provide class hierarchies for the Scheduler and RecoveryManager so that the DataManager is able to choose the most adequate components for the data object. The programmer may define its own Scheduler or RecoveryManager by subclassing from the existing class hierarchies.

We may create different DataManagers by replacing the existing Scheduler and/or RecoveryManager. Note that the Scheduler and RecoveryManager are independent of each other. Replacing one of these components will have no effect on the other.

4.4  Introducing dynamic behavior

Now that we have identified the different components of the transaction system we wish to introduce dynamic behavior in our system. With dynamic behavior we mean the ability to switch between different transaction semantics. Figure 4.6 shows an example of a transaction system. In this figure the semantics for different components are chosen.
In our transaction system we shall only tune between different concurrency control methods, i.e. schedulers for the handling of the messages at the data objects. In figure 4.6 a user specifies its desired scheduler by using one of the classes of the TransactionManager class hierarchy. He may also define its own kind of TransactionManager by subclassing from one of the classes of the TransactionManager class hierarchy.

Each TransactionManager in the TransactionManager class hierarchy should differ from the other in that it chooses a different scheduler. The TransactionManager class hierarchy can have the following category of TransactionManagers:

- A TransactionManager that doesn’t choose a scheduler at all.
- Static TransactionManagers that choose a specific scheduler
- A dynamic TransactionManager that selects an adequate scheduler for the application according to a
Chapter 4  Design of the framework

predefined algorithm.

By inheriting from one of these TransactionManagers the user tells the transaction system that he wishes its operations to be handled by the chosen scheduler. The TransactionManager will pass the chosen scheduler to the PolicyManager to denote the user’s wish.

The system self must also be able to denote its choice for a kind of scheduler. Therefore, The PolicyManager class hierarchy represents classes that choose different schedulers. This can be categorized in the same way as the TransactionManager class hierarchy. This means that we have a PolicyManager that doesn’t chooses any scheduler, PolicyManagers which choose a specific scheduler and a PolicyManager which selects a scheduler according to an algorithm.

In our transaction system both the user and the PolicyManager are able to choose a scheduler for the data operations. We have to define a priority mechanism to determine which scheduler we must choose that must be passed to the DataManager of the data object. If only one of the two chooses a scheduler then we decide to pass that scheduler to the DataManager. If none of the two parties choose a specific scheduler then we pass an empty scheduler to the DataManager meaning that the DataManager itself should choose a scheduler. If both the TransactionManager and the PolicyManager choose a scheduler then we have to decide which of the two chosen schedulers must be passed to the DataManager. In our framework we have decided to give the user higher priority than the PolicyManager, that is, if the user has chosen a scheduler then this will always be submitted to the DataManager. Of course it is also possible to give the PolicyManager higher priority. Note however that the TransactionManager and the PolicyManager may adopt different criteria for the choice of schedulers. The chosen scheduler by the TransactionManager may for example be based on the kind of the application, whereas the PolicyManager may use environment conditions like the performance to choose a scheduler. The four different situations of choosing schedulers by the TransactionManager (TM) and the PolicyManager (PM) are represented in figure 4.7:

<table>
<thead>
<tr>
<th>TM</th>
<th>PM</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

Figure 4.7  Four possible situations in choosing schedulers by the TM and the PM
The DataManager of the data object will receive messages from transactions together with the kind of scheduler the message is desired to be scheduled by. If the DataManager is already using a scheduler, that is, if it is busy with a transaction, then the active scheduler must be used. However, if the scheduler of the DataManager is idle, which means that the DataManager isn’t involved in an access of any transaction, then the scheduler chosen by the system, i.e. the TransactionManager or the PolicyManager, will be used by the DataManager. If meanwhile other transaction messages will be received by the DataManager then they will also be handled by the same active scheduler. The DataManager chooses its default scheduler if the system hasn’t chosen a scheduler and if the DataManager was idle. The scheduler is reactivated if the DataManager has terminated (committed or aborted) all the transactions that has accessed the data object. This whole priority mechanism is shown in figure 4.8.

<table>
<thead>
<tr>
<th>System</th>
<th>Scheduler active</th>
<th>DM chooses</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>yes</td>
<td>active</td>
</tr>
<tr>
<td>yes</td>
<td>no</td>
<td>argument</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>argument</td>
</tr>
<tr>
<td>no</td>
<td>no</td>
<td>default</td>
</tr>
</tbody>
</table>

Figure 4.8  The priority mechanism for choosing schedulers

Now that we have described all the components of the framework and defined a priority mechanism for dynamically associating different schedulers to different objects we can build our framework. The implementation of the class hierarchies are given in the next chapter.
Chapter 5

Implementation of the Framework

Abstract

This chapter describes the implementation of the framework. A model of the transaction system will be given. Additionally the main classes and class hierarchies of the framework will be described.


5.1 Introduction

In this chapter the object oriented atomic transactions framework model will be described. The structure of our framework is illustrated in the model of figure 5.1. In this model each module sends requests to and receives replies from the next lower level module. Transactions submit their operations to their TransactionManager. Operations can be ordinary user messages or one of the transaction messages \textit{startTransaction}, \textit{commit} and \textit{abort}. Each transaction has one TransactionManager. The TransactionManager performs any required preprocessing of the operations it receives from its transaction. Subsequently, the TransactionManager submits the operations to the PolicyManager which is responsible for timestamp initiating and passing the operations to the intended objects called DMOObjects. The PolicyManager is shared by several TransactionManagers.

![Diagram of the transaction system]

**Figure 5.1** Model of the transaction system

When the DMOObject receives an operation from the PolicyManager it passes these over to its DataManager, which is responsible for consistent access of the state of the object. The DataManager contains a Scheduler which controls the relative order in which operations are executed. The Scheduler receives read, write, commit and abort operations from the DataManager. It can decide to accept, delay or reject the received operation. If the Scheduler rejects the operation, it sends a negative
acknowledge back to the DataManager. This acknowledge will pass the PolicyManager, TransactionManager and will be finally submitted to the Transaction. The Transaction invokes then an abort message. If the Scheduler accepts the received message then it sends a positive acknowledge to the DataManager which will eventually result in a positive acknowledge to the Transaction. If the Scheduler accepted the operation then the operation is send to the RecoveryManager which processes it by manipulating the state of the object. In case of a read message the acknowledge will include the value read.

In the following sections the main classes of the framework will be presented. Here, only the definitions of the classes will be handled. Implementation details are only given when it is necessary. Appendix A presents the Sina implementation of our framework. Appendix B contains the Smalltalk implementation.

### 5.2 Class ApplicationWithTransaction

The class ApplicationWithTransaction is used for initiating transactions. Its definition is given in figure 5.2. This class inherits from the TransactionManager class which will be handled in the next section.

```smalltalk
class ApplicationWithTransaction
  interface
    internals
      tm: TransactionManager;
    methods:
      transmethod returns Nil;
      commit returns Nil;
      abort returns Nil;
    inputfilters
disp: Dispatch = { True => inner.* }
  outputfilters
    constr: Meta = {tm.Busy => [.*,] tm.handleMessage}
  end

class ApplicationWithTransaction
  implementation
  methods
    transmethod
      begin
        self.startTransaction
        ...
        self.commit / self.abort
      end;
    .....
  end;

Figure 5.2 Definition of the class ApplicationWithTransaction
```
The ApplicationWithTransaction class has no instance variables. It has the methods transmethod, commit and abort.

The method transmethod is used for creating transactions. The programmer must use the inherited method startTransaction to initiate its transaction. The optional methods commit and abort in the class ApplicationWithTransaction are used to define the user actions in case of a commit and abort, respectively. If the user doesn’t want to implement any user actions in case of a commit or abort the methods commit and abort of the inherited TransactionManager will be invoked.

The class ApplicationWithTransaction uses an inputfilter disp of the class Dispatch to inherit the behavior of the class TransactionManager. All the messages used in the method transmethod will be submitted to the outputfilter constr, if they are outside of the object’s boundaries and if the condition Busy of the class TransactionManager is true. The condition Busy of the class TransactionManager is set to true if startTransaction has been invoked but not the methods commit or abort, that is, if the application object is involved in a transaction.

The outputfilter constr converts the received messages to their first-class representation, i.e. to an instance of class Message. The reified message will then be passed as an argument of the TransactionManager method handleMessage to the ACT class TransactionManager.

### 5.3 Class TransactionManager

The class TransactionManager is the class which provides the transaction operations to the class ApplicationWithTransaction. It’s definition is given in figure 5.3. This class inherits from the class SenderCommitProtocol. The class SenderCommitprotocol provides the commit protocol for the TransactionManager.

The method startTransaction may only be invoked if the condition Busy is false. If the condition Busy is true then all the messages except startTransaction may be invoked. This construction guarantees a correct syntax of the initiated transaction. The method startTransaction will set the boolean variable busyTrue to true and additionally it will invoke the message handleStart of the shared class PolicyManager. The PolicyManager on its turn will invoke the message setTimestamp of the class TransactionManager, which will result in the initialization of the timestamp of the initiated transaction. Note that this class supports two different views. One view is for the ApplicationWithTransaction class who may invoke all the messages except setTimestamp. The other view is for the system (PolicyManager) who may only invoke the message setTimestamp.
The method `handleMessage` sends the reified message together with the timestamp to the class `PolicyManager` which will provide the subsequent handling of the message.

The `TransactionManager` keeps a list of all the objects it has sent messages to. In case of a commit or abort the `TransactionManager` must send commit or abort respectively to all the objects that the `TransactionManager` accessed. This process is done by the `SenderCommitProtocol` class. The commit of a transaction is done by applying the two-phase commit protocol.

```plaintext
class TransactionManager interface
  internals
    senderCommit: SenderCommitProtocol;
  conditions
    application_view;
    system_view;
  methods
    startTransaction returns Nil;
    handleMessage(Message) returns Nil;
    commit returns Nil;
    abort returns Nil;
    setTimestamp (Integer) returns Nil;
  conditions
    Busy;
  inputfilters
    error: Error = { application_view => *
                      setTimestamp,
                      system_view => setTimestamp ];
    disp: Dispatch = { Busy.not => inner.startTransaction;
                      Busy => inner.*, senderCommit.* \ inner.startTransaction];
end;
```

**Figure 5.3** Definition of the class `TransactionManager`

### 5.4 Class PolicyManager

The class `PolicyManager` is shared by different `TransactionManager` class objects. It provides timestamps for its clients. Moreover, it submits the actual transaction messages to the corresponding objects. Furthermore, it initializes an adequate scheduler for the transaction. The definition of the class `PolicyManager` is illustrated in figure 5.4.

The class `PolicyManager` has an instance variable called `timestampCounter`, which is unique in the whole system. This variable is an instance of the class `Association`. The key of the Association represents the global id of the `PolicyManager` object in the system. This part is set at instance creation time of the `PolicyManager` (by the initial method). The value of the Association is the local timestamp.
counter (see figure 5.5).

```plaintext
class PolicyManager(globalId: Integer)  interface
  methods
    handleStart  returns Nil;
    setTimestampCounter  returns Nil;
    handleOperation(Message, Integer)  returns Boolean;
    chooseScheduler  returns UniversalScheduler;
  conditions
    setStampsNotBusy;
  inputfilters
    mutex: Wait = { setStampsNotBusy => *.setStamps};
    disp: Dispatch = {True => inner.*};
end;
```

**Figure 5.4** The definition of the PolicyManager

Each time when a startTransaction message is received the message `self.setTimestampCounter` is invoked. This results in the updating of the instance variable timestampCounter. In our implementation the value part of the timestampCounter is just incremented by one. Since several TransactionManager objects may invoke the message startTransaction concurrently, the message setTimestamp must be mutually excluded to provide unique timestamps.

![global id counter](image)

**Figure 5.5** The representation of the instance variable timestampCounter

After a correct updating of the timestampCounter the sender of the `startTransactionMessage` is then send the new timestamp by invoking the message `sender.setTimestamp`.

The whole process of initiating a transaction is illustrated in figure 5.6. In this figure.awt, tm and PM are instances of the classes ApplicationWithTransaction, TransactionManager and PolicyManager respectively. The brackets “<“ and “>” in step 4 are used to denote that this message will be invoked atomically.
The implementation of the method `handleOperation` is illustrated in figure 5.7.

```plaintext
handleOperation(aMessage:Message, aTimestamp:Integer)
  temp target: Object;
  result: Boolean;
  begin
    target:=aMessage.getObject;
    result:=target.handleMessage(aMessage, aTimestamp, server.choosescheduler);
  return result
end;
```

**Figure 5.7** The implementation of the method `handleOperation`

This method gets the target from `aMessage` and invokes the message `handleMessage` on it. Moreover, it initializes the desired scheduler, by invoking `server.chooseScheduler`. The message `server.chooseScheduler` will be first submitted to the `TransactionManager`. If the `TransactionManager` defines such a message then that message will be invoked. Otherwise the message `chooseScheduler` of the class `PolicyManager` will be invoked. The `chooseScheduler` method returns an instance of a subclass of the `UniversalScheduler` class, which is the superclass of all the schedulers in the framework. The method `chooseScheduler` chooses an adequate scheduler based on the user requirements and the actual state of the system.

The `TransactionManager` of figure 5.3 doesn’t define a `chooseScheduler` method meaning that the user doesn’t care which scheduler will be used for its transaction operations. However if the user wishes its operations to be scheduled by a specific scheduler he can make a subclass of the `TransactionManager` class which does provide a `chooseScheduler` method. Figure 5.8 illustrates the class `TwoPLTransactionManager` which provides a `chooseScheduler` method that returns a `TwoPLScheduler` object.
Chapter 5  Implementation of the framework

class TwoPLTransactionManager interface
  internals
    tm: TransactionManager;
  methods
    chooseScheduler returns UniversalScheduler;
  inputfilters
    disp: Dispatch = { True => tm.*, inner.* }; end;

Figure 5.8  The definition of the TwoPLTransactionManager class

Our framework provides for each (concrete) Scheduler class a corresponding TransactionManager class. The question arises here if it isn’t possible that the scheduler can be chosen dynamically by the chooseScheduler method. The framework provides a solution for this problem also. The class DynamicTransactionManager chooses a scheduler based on the gathered information about the transaction that is being handled by the DynamicTransactionManager. This information gathering can be done during the processing of the transaction. The DynamicTransactionManager can for instance decide not to choose an optimistic scheduler when it notices that it is dealing with a long transaction (optimistic schedulers show bad performance for long transactions). Figure 5.9 represents the class hierarchy for TransactionManagers.

Figure 5.9  The definition of the class DynamicTransactionManager

5.5 Class DataManager

Each data object in our transaction system inherits from the class DataManager. The definition of a data object is given in figure 5.10.
class DataManagerObject interface
    internals
        dm: DataManager;
        ....
    inputfilters
        disp: Dispatch = [ True => dm.*, inner.*];
end;

Figure 5.10 Definition of the DataManagerObject

The class DataManager is responsible for the consistent access to its server object. It has as internals the classes UniversalScheduler and RecoveryManager. The definition of the DataManager class is given in figure 5.11.

class DataManager interface
    internals
        scheduler: UniversalScheduler
        recoveryManager: RecoveryManager;
    methods
        handleMessage (Message, Integer, UniversalScheduler) returns Association;
        handleCommitRequest(Integer) returns Nil;
        handleAbort returns Nil;
        handleCommit returns Nil;
    inputfilters
        disp: Dispatch = [ True => scheduler.*, recoveryManager.*, inner.*]
end;

Figure 5.11 Definition of the DataManager class

The method handleMessage returns an Association object which consists of a boolean key and the value of a read message. Initially both values are nil. The key part of the Association object is filled by the scheduler which decides if the received message can be scheduled or not. The key part of the Association can have the following values:

• true; The scheduler has decided that the received message can be scheduled. The DataManager sets the key of the Association to true and subsequently it sends the received message to the recoveryManager.
• false; The scheduler has decided that the received message can not be scheduled. The DataManager sets the key of the Association to false but the message is not send to the recoveryManager.
• nil; The scheduler made the decision to schedule the received message. However, the message should not be send to the recoveryManager. This value is returned by the ThomasWrite2PLScheduler which will be handled later.
The value part of the Association will be filled by the recoveryManager which is responsible for performing the message. In addition, the recoveryManager does some bookkeeping for recovery in case of failures. The value part has the value of a read message or Nil in case of a write message.

The method handleMessage also decides which scheduler has to be chosen for the received message. The priority mechanism is implemented as described in chapter 4 about the design of the framework.

The method handleCommitRequest is invoked by a transaction which wishes to commit. The DataManager submits this commit request to its scheduler which decides whether the transaction can be committed. The methods handleAbort and handleCommit inform the scheduler and recoverymanager that the sender of these messages have aborted or committed respectively.

### 5.6 Class UniversalScheduler

A scheduler controls the concurrent execution of transactions. It exercises this control by restricting the order in which the DataManager executes the transaction messages abort and commit and data messages. Its goal is to order these operations so that the resulting execution confirms to the correctness criteria. As we have seen before one of these correctness criteria is serializability. In addition to providing serializable executions the scheduler has also the task of creating at least recoverable executions. It may also ensure that the executions avoid cascading aborts or are strict. In our framework all the schedulers produce strict and serializable executions.

After receiving a message the scheduler can take one of the following three actions:

1. **Accept:** The message can be executed. The DataManager acknowledges the execution of the message to the sender.
2. **Reject:** The scheduler sends a negative acknowledge back to the DataManager, which on his turn informs the sender of the message.
3. **Delay:** The scheduler can delay the message by placing it in a queue internal to the scheduler. Later it can remove it from the queue and either execute or reject it. Meanwhile the scheduler is free to schedule other operations.

Before performing one of these actions the scheduler must take a decision. This decision describes the behavior of a scheduler.

The UniversalScheduler class as shown in figure 5.12 provides the abstract base class providing the
framework for the scheduling system in our transaction system. Each scheduler subclass provides a concrete implementation of a particular scheduling policy. The UniversalScheduler class is itself only an abstract definition of the components of the class and thus cannot have an instance. However, this top-level of the scheduler hierarchy needs to be viewed to understand how the subclass implementations fit into the transaction system. The subclasses of the UniversalScheduler class may redefine the methods according to their specific needs.

```plaintext
class UniversalScheduler interface
  methods
    // scheduler dependent methods.
    abort returns Nil;
    commit returns Nil;
    commitRequest returns Boolean;
    readdecide(Any, Integer) returns Boolean;
    writedecide(Any, Integer) returns Boolean;
    // implemented methods, reused by all schedulers.
    decide(Message, Integer) returns Boolean;
    isActive returns Boolean;
    reActivate returns Boolean;
    setActive returns Nil;
  inputfilters
    disp:Dispatch = [True => inner.*]
end
```

**Figure 5.12** The definition of the UniversalScheduler class

The UniversalScheduler class has one instance variable called `active`, which is accessed by the methods `isActive`, `reActivate` and `setActive`. The method `isActive` tests if the corresponding scheduler is active. The methods `reActivate` and `setActive` are used for reactivating and activating the scheduler. These methods are used in the DataManager method `handleMessage`, which has to decide which scheduler to choose.

The method `decide` is also reused without reimplementing by all the schedulers. The implementation of this method is shown in figure 5.13.

```plaintext
decide(aMessage: Message, timestamp: Integer)
begin
  if aMessage.isRead
  then return self.readdecide(aMessage, timestamp)
  else return self.writedecide(aMessage, timestamp)
end;
```

**Figure 5.13** The implementation of the decide method
This method is invoked by the DataManager (in the `handleMessage` method). It checks whether the received message is a read or write and invokes then the message `self.readdecide` or `self.writedecide`. This method is a polymorphic method because depending on the kind of scheduler its behavior is different. The methods `commit` and `abort` are also scheduler specific and thus are unimplemented in the `UniversalScheduler` class. The method `commitRequest` just returns true. This method is invoked by the class `SenderCommitProtocol` which sends commit requests to all the objects that its `TransactionManager` accessed. In our framework only the optimistic schedulers reimplement this method.

Figure 5.14 illustrates a graphical representation of the scheduler class hierarchy in the framework. The arrows denote inheritance. Figure 5.15 illustrates the class hierarchy for the implemented schedulers in a table. The arrows in the table indicate inheritance here also. Inheritance permits similar but new schedulers to be added easily. Much of the code from one scheduler will be the same as in another scheduler and can be easily inherited. As an example, the `TWRTOScheduler` class only needs to re-implement the `writedecide` method it inherited from the `TOScheduler` class.

The methods `isActive`, `setActive`, `reActive` and `decide` are not included in the table of figure 4.15 since these methods are general enough that they can be inherited by all the scheduler classes without any modifications.

![Scheduler Class Hierarchy Diagram](image-url)
5.7 Locking scheduler classes

Our framework provides different locking scheduler classes. The class Aggressive2PLScheduler which represents the two-phase locking scheduler is the superclass of all the locking scheduler classes. The definition of this class is illustrated in figure 5.16:

This class has a LockManager as an internal. The lockManager services the readlock and writelock messages which set a read lock and write lock respectively on the corresponding object. When the scheduler receives a read message or write message from the DataManager it sends the appropriate lock operation to the LockManager. The LockManager then checks if the lock can be acquired or not and then subsequently sends a positive or negative acknowledge to the scheduler. The scheduler can do two things in this case: either reject the operation or try again until the lock has been set. The Aggressive2PLScheduler class rejects each operation if it can’t be locked at the first time. Thus, it never
delays operations. The methods commit and abort invoke the LockManager’s removelocks message to release all the locks of a transaction.

class Aggressive2PLScheduler interface
    internals
        us: UniversalScheduler;
        lockManager: LockManager;
    methods
        readdecide(Message, Integer) returns Boolean;
        writedecide(Message, Integer) returns Boolean;
        commit(Message) returns Nil;
        abort(Message) returns Nil;
    inputfilters
        disp: Dispatch = [ True => us.*, lockManager.*, inner.*]}
end;

Figure 5.16 Definition of the Aggressive2PLScheduler class

The opposite of the Aggressive2PLScheduler is the class Basic2PLScheduler which delays operations if a lock cannot be acquired. Its definition is given in figure 5.17. The Basic2PLScheduler class inherits from the Aggressive2PLScheduler class. It needs only to reimplement the readdecide and writedecide methods. In these methods the Basic2PLScheduler class tries to set a required lock until it is granted. An important and unfortunate property of the Basic2PLScheduler class is that it is subject to deadlocks. In order to detect and to solve deadlocks the readdecide and writedecide methods of the Basic2PLScheduler class use the additional methods detectDeadlock and solveDeadlock. These two methods are not implemented in this class. As a result the Basic2PLScheduler class is only partially implemented and therefore is an abstract class (and thus may not have any instances).

class Basic2PLScheduler interface
    internals
        aggressive2PLScheduler: Aggressive2PLScheduler;
        distributedDeadlockDetector: DistributedDeadlockDetector;
    methods
        checkforDeadlock(Any, Integer, Any, Integer) returns Nil;
        solveDeadlock(Any, Integer, Any, Integer) returns Nil;
        readdecide(Message, Integer) returns Boolean;
        writedecide(Message, Integer) returns Boolean;
    inputfilters
        disp: Dispatch = [ True => aggressive2PLScheduler.*, distributedDeadlockDetector.*, inner.* ];
end;

Figure 5.17 The definition of class Basic2PLScheduler

The Basic2PLScheduler’s subclasses are responsible for implementing the methods detectDeadlock and solveDeadlock. Depending on the kind of deadlock resolution techniques these methods may be imple-
mented differently. In our framework there are four subclasses of the Basic2PLScheduler class. The WFG2PLScheduler class represents the Wait-For-Graph deadlock detection mechanism. The classes WW2PLScheduler and WD2PLScheduler implement respectively the Wound-Wait and Wait-Die deadlock prevention mechanisms. Finally, the Timeout2PLScheduler class is based on a time-out mechanism to solve deadlocks at the object. Note that the readdecide and writedecide methods are inherited by the Basic2PLScheduler’s subclasses without any modification.

To cope with distributed deadlocks our framework provides the DistributedDeadlockDetector class. The readdecide and writedecide methods first try to set the appropriate lock. If the lock cannot be granted then it is checked if there is a local deadlock. If this is the case then the detected local deadlock will be solved by aborting one of the deadlock participants. The deadlock may be distributed. Therefore, the methods readdecide and writedecide ask the DistributedDeadLockDetector class to detect and solve a possible recent distributed deadlock.

This DistributedDeadLockDetector class is an internal of the Basic2PLScheduler class. This class may be based on different protocols to solve the distributed deadlock. The distributed deadlock may be solved centrally or by inter-object communication between different DistributedDeadlockDetector classes. In our framework only the centralized approach has been implemented. The system provides a GlobalDeadlockDetector class which is based on the Wait-for-Graph deadlock-detection mechanism. If a local scheduler suspects that a transaction is involved in a distributed deadlock, then it informs the global deadlock detector which on his turn detect and solve the occurred deadlocks.

5.8 Timestamp ordering scheduler classes

The TOScheduler class represents the timestamp ordering scheduler. Its definition is illustrated in figure 5.18. The TOScheduler class inherits directly from the UniversalScheduler class and overrides the methods readdecide, writedecide, commit and abort. This class uses two instance variables called readTimestamps and writeTimestamp providing a set of the latest readers and writer timestamps of the object. Initially readTimestamps is an empty set, whereas writeTimestamp is assigned the integer value 0. The methods readdecide and writedecide implement the timestamp ordering message handling. The methods commit and abort remove the timestamp of the terminated transaction from readTimestamps and writetimestamp.
The TWRTOScheduler in the framework represents the Thomas-Write Timestamp Ordering scheduler. As we have seen before this scheduler differs from the basic timestamp ordering scheduler in that it ignores late write messages. Thus this class can inherit from the TOScheduler class whereby it needs only to redefine the method writedecide. See figure 5.19 for the definition of this class.

5.9 Optimistic scheduler classes

The abstract superclass of the optimistic schedulers in our framework is the class OptimisticScheduler. This class inherits directly from the UniversalScheduler class and redefines the methods readdecide, writedecide, commit, abort and commitRequest. The method commitRequest is implemented by invoking the message self.subclassResponsibility thereby denoting that this method should be implemented by the subclasses. The OptimisticScheduler class uses two instance variables called readDictionary and writeDictionary which are instances of the system class Dictionary. The key of each Association in these directories contain the sender of the message, whereas the value contains the timestamp of the sender. The readdecide and writedecide methods only save the message with its timestamp in the readDictionary or writeDictionary depending on the kind of message. Moreover they always return true meaning that the received message is accepted. The methods commit and abort remove the sender transaction from
the readDictionary and writeDictionary. See figure 5.20 for the definition of the OptimisticScheduler class.

```plaintext
class OptimisticScheduler interface
    internals
        us: UniversalScheduler;
    methods
        readdecide(Message, Integer) returns Boolean;
        writeddecide(Message, Integer) returns Boolean;
        commit(Message) returns Nil;
        abort(Message) returns Nil;
        commitRequest returns Boolean;
    inputfilters
        disp: Dispatch = { True => us.*, inner.*};
end;
```

Figure 5.20 The definition of the OptimisticScheduler class.

The OptimisticScheduler class has two subclasses called OptimisticTOScheduler and Optimistic2PLScheduler. These classes only redefine the method commitRequest of the OptimisticScheduler class. The OptimisticTOScheduler class is based on timestamp ordering certification. The Optimistic2PLScheduler class adopts a two-phase locking certification.

5.10 Class SerialScheduler

The class SerialScheduler represents a very trivial scheduler, namely a scheduler which processes only one transaction at a time. So if it has received a message from a transaction then it denotes this transaction as its client and rejects all the other transaction messages until the client transaction terminates. The instance variable client is used to denote the transaction that is active at this scheduler. When the client transaction terminates, i.e. when the serial scheduler receives an abort or commit from the client transaction, then the instance variable is set to nil meaning that the scheduler is idle. Figure 5.21 illustrates the definition of the class SerialScheduler.
class SerialScheduler interface
  internal
  us: UniversalScheduler;
  methods
    readdecide(Message, Integer) returns Boolean;
    writedecide(Message, Integer) returns Boolean;
    commit(Message) returns Nil;
    abort(Message) returns Nil;
  inputfilters
    disp: Dispatch = [True => us.*, inner.*];
end;

Figure 5.21  The definition of the class SerialScheduler

5.11 Class RecoveryManager

If the scheduler accepts a received message then the DataManager sends this message to the RecoveryManager which actually performs the message. In our framework the RecoveryManager is only implemented to deal with transaction failures. The definition of the RecoveryManager class is given in figure 5.22. The RecoveryManager class has one instance variable called, aftervalue. The aftervalue variable represents the state of the object the RecoveryManager belongs to. If the RecoveryManager receives a read message it reads from the aftervalue if available, otherwise if the aftervalue is nil then it reads from the original object. When the RecoveryManager receives a write message it assigns the state of its server object to aftervalue.

class RecoveryManager interface
  methods
    abort(Any) returns Nil;
    commit(Any) returns Nil;
    handleMessage(Message) returns Any;
  inputfilters
    disp: Dispatch = [True => inner.*];
end;

Figure 5.22  The definition of the RecoveryManager class.
Evaluation & Conclusions

We have developed an object-oriented framework for atomic transactions. This framework is implemented in Smalltalk and Sina. Since at the writing of this thesis the Sina compiler was not totally implemented yet, the framework could only be tested with Smalltalk.

The design of the framework addresses the problems mentioned in chapter 4. The transaction system is decomposed into well-defined subcomponents which encapsulate different transaction semantics. Within the framework different transaction protocols are organized in an inheritance hierarchy. The programmer can use objects of these classes to construct its own transaction system. Moreover, he can tailor from the existing transaction semantics by subclassing from the classes in the class hierarchies to define its own system classes.

The transaction framework also shows a high degree of flexibility. According to the conditions different transaction semantics can be selected at an object-level. This selection can be done at three levels: the application level, the environment level and the data level. Each of these may require different transaction semantics to be adopted. A priority mechanism is developed to solve the requirements for different transaction semantics among the three parties.

In order to improve the constructed framework the following additional work should be done:

• The framework has only one RecoveryManager class. This should be extended to a class hierarchy.
• The scheduler class hierarchy is based on a syntactic concurrency control model using the serializability theory. Also concurrency control models using semantic information should be studied to include in the framework.
• Performance study about concurrency control methods is needed to define the criteria for switching between different schedulers.
Appendix A

Sina sources
Appendix B

Smalltalk sources
References


References

[Campbell 91] R.H. Campbell, G.M. Johnston, P.W. Madany, V.F. Russo, 
Principles of Object-Oriented Operating System Design, Dept. of Computer Science, 
University of Illinois at Urbana-Champaign, 1991.

[Cellary 89] W. Cellary, E. Gelenbe & T. Morzy, Concurrency Control in Distributed Database Systems, 


[Date 83] C.J. Date, An Introduction to Database Systems, Vol. 3, 
Addison Wesley, 1990.

[Elmagarmid 91] A.K. Elmagarmid, editor, Transaction Management in Database Systems, 

[Garcia-Molina 83] H. Garcia-Molina, Using Semantic Knowledge for transaction processing in a 
distributed database, ACM Transactions on Database Systems, 8(2) :186-213, June 1983.

[Goldberg 89] A. Goldberg & D. Robson, Smalltalk-80, The Language and its Implementation, 
Addison Wesley, 1989.


[Hadzilacos 88] V. Hadzilacos, A theory of reliability in Database Systems, 

[Haerder 83] T. Haerder & A. Reuter, Principles of Transaction-Oriented Database Recovery, 

[Johnson 88] R.E. Johnson & B. Foote, Designing Reusable classes, 
JOOP, June/July 1988, pp. 22-35.

[Knapp 87] E. Knapp, Deadlock detection in Distributed Databases, 

[Kung 81] H.T. Kung & J.T. Robinson. On optimistic Methods for Concurrency Control, 
References


Index

A
abort 5
abstract class 38
abstraction 39
ACID properties 5
ACT 55
analysis phase 41
Application object 58, 63
associative inheritance 58
atomic transaction 5
Atomicity 5
ACA (avoiding cascading aborts) executions 26

C
centralized 2PC 31
centralized systems 4
checkpointing 30
class 3
class ApplicationWithTransaction 72
class DataManager 77
class PolicyManager 73
class RecoveryManager 87
class SerialScheduler 86
class TransactionManager 73
class UniversalScheduler 79
commit 5
composition filters 1
    model 47
concurrency
    control 5, 10
    models 11
    problem 5
conflict 12
consistency 5
coordinated behavior 46
correct executions 33

D
Data object 65
DataManager 6, 66, 71
deadlock 16
    avoidance 18
    detection 18
    prevention 17
    resolution methods 17
decentralized 2PC 31
delegation 40, 52
design phase 41
distributed 4
deadlock 17
Index

optimistic concurrency control 23
system 4, 25
timestamp ordering 22
two-phase locking 16
durability 6
dynamic behavior 66

E
early decomposition 44
encapsulation 37
execution atomicity 6
extensibility 37

F
failure atomicity 6
fault tolerance 4
framework 7

G
granularity 19

I
implementation phase 41
inconsistent retrieval 10
information hiding 37
inheritance 39, 52
in-place-updating 29
input filters 49
instance 38
intention read lock 19
intention write lock 19
isolation 6
iteration model 41

L
large software systems 37
linear 2PC 31
local deadlock 17
lock type graph 19
locking 15
logging 29, 30
lost update 10

M
maintainability 37
media failures 25
message 38
method 38
multiple granularity locking 18
multiple views 46, 57

O
object 38
object model 39
object-oriented concepts 1
Index

optimistic concurrency control 15, 23
optimistic scheduler classes 85
outputfilters 55, 56

P
parallelism 4
PolicyManager 64, 71
priority mechanism 69

R
readlock 15
recoverable executions 26
recovery 25
  problem 5
  protocols 6
RecoveryManager 6, 66, 71
reusability 37

S
Scheduler 6, 66, 71
Scheduler class hierarchy 82
semantic concurrency control model 12
serial execution 11
serial execution example 12, 13
serializability 12
serializable execution example 13, 14
serialization graph 14
shadowing 29
Sina 1
Smalltalk 1
startTransaction 5
strict executions 27
syntactic concurrency control model 12
system failures 25

T
timestamp management 21
timestamp ordering 15, 21
timestamp ordering schedulers 84
transaction 4
transaction failure 26
transaction system model 71
TransactionManager 6, 63, 71
transparency 4
Trese 1
two phase commit 30
two phase locking 15

W
wait die 18
waterfall model 41
wound wait 18
write-ahead logging 29
writelock 15