

Twin-Transaction Model To Support Mobile Data Access

Aamir Rasheed

Bachelor of Mathematics
Punjab Univerisy, Lahore, Pakistan.

Master of Computer Science
Quad-i-Azam University, Islamabad, Pakistan.

Master of Computer Science
Asian Institute of Technology, Bangkok, Thailand.

School of Computer Science and Software Engineering,

Monash University,

Australia

December 1999

Contents

CONTENTS	III
ABSTRACT	IX
STATEMENT	XI
ACKNOWLEDGMENTS	XIII
LIST OF FIGURES	XV
LIST OF TABLES	XVII
PUBLICATIONS	XIX
CHAPTER 1 INTRODUCTION	1
1.1 Research Background	1
1.2 Mobile Computing	4
1.2.1 Disconnected Operation and Data Inconsistency	5
1.3 Motivation and Objectives	7
1.4 Twin-Transaction Model	9
1.5 Dissertation Outline	10
CHAPTER 2 BACKGROUND	13
2.1 Mobile Computing	13
2.2 Traditional Database Systems	18
2.2.1 Concurrency Control	20
2.2.2 Recovery	22

2.2.3	Impact of Mobile Computing Environment.....	24
2.3	Advanced Transaction Models	27
2.3.1	Sagas.....	28
2.3.2	Multilevel Transactions.....	30
2.3.3	Split Transactions.....	31
2.3.4	S Transaction	33
2.3.5	Flex Transactions.....	34
2.3.6	DOM Transactions	35
2.3.7	Cooperative Transaction Model.....	37
2.3.8	ConTract Model.....	38
2.4	Replication.....	40
2.4.1	Synchronous Replication Protocols.....	41
2.4.2	Asynchronous Replication Protocols	47
2.5	Related Research Issues	57
2.5.1	Weak Operations and Weak Consistency	57
2.5.2	Currency Tokens.....	59
2.5.3	Coda	60
2.5.4	Isolation-Only Transactions.....	61
2.5.5	Bayou.....	62
2.5.6	Kangaroo Transactions	65
2.6	Summary.....	68
CHAPTER 3 TWIN-TRANSACTION MODEL.....		71
3.1	Design Decisions	71
3.1.1	Disconnected Operation.....	74
3.1.2	Inconsistency Detection and Correctness Criteria	75

3.2	The Model	76
3.2.1	Transaction Management	78
3.2.2	Data Item Structure.....	79
3.2.3	Transaction Structure.....	80
3.3	Resynchronisation	82
3.4	Consistency Model.....	83
3.4.1	Local Consistency.....	84
3.4.2	Global Consistency	84
3.5	Conflict Resolution	86
3.5.1	Re-execution.....	87
3.5.2	Abort.....	89
3.5.3	Application Specific.....	90
3.5.4	Human Intervention.....	92
3.5.5	No-Conflict.....	92
3.6	ACID Properties.....	92
3.7	Twin Transaction Specification.....	94
3.8	Summary.....	95
CHAPTER 4 TRANSACTION EXECUTION AND MANAGEMENT.....		97
4.1	Transaction Execution.....	97
4.2	Transaction History.....	100
4.2.1	Intelligent Log Maintenance	106
4.3	Concurrency Control	109
4.3.1	Connected Mode Operation	110
4.3.2	Local Concurrency Control.....	115

4.4	Disconnected Mode Operation	117
4.4.1	Probabilistic Conflict Detection	117
4.4.2	Transaction Execution Rules and Patterns.....	121
4.5	Summary.....	127
CHAPTER 5 RESYNCHRONISATION		129
5.1	TTM State Propagation	129
5.1.1	FTM to MTM	131
5.1.2	MTM to FTM	133
5.2	Validation Process	135
5.3	Data Item States	136
5.4	Transaction Resolution	140
5.4.1	Transaction Resolution Site.....	145
5.4.2	Safety and Acceptance of Transaction Resolution	146
5.4.3	Application Specific and Human Intervention Resolution Issues.....	148
5.5	Summary.....	150
CHAPTER 6 IMPLEMENTATION		151
6.1	Architecture	151
6.2	Data Structures	153
6.2.1	Transaction Lists.....	154
6.2.2	Precedence Graph	155
6.2.3	Wait-for-Graph	155
6.2.4	π -Database	155
6.2.5	Miscellaneous Data.....	156
6.2.6	Twin-Transaction Structure	156

6.2.7	Data Item Node (DI-Node)	158
6.3	Maintaining Data Structures	159
6.3.1	Transaction Lists.....	159
6.3.2	Read and Write Set Maintenance.....	159
6.4	Optimisation	160
6.4.1	DI-Node Space Management.....	160
6.4.2	$\pi(T)$ Management	161
6.5	Persistence and Crash Recovery.....	162
6.5.1	Persistent Data Structures	162
6.5.2	Crash Recovery	163
6.6	Resynchronisation Revisited	166
6.6.1	Data Item Maintenance	166
6.6.2	Transaction Synchronisation.....	169
6.7	Transaction Script.....	171
6.8	Application Specific Resolvers	173
6.9	Summary.....	174
CHAPTER 7	EVALUATION	175
7.1	Overview/System Status.....	175
7.2	Evaluation Approach	177
7.3	Transaction Performance	178
7.3.1	Methodology	178
7.3.2	Results.....	180
7.4	Resource Cost Measurements	186
7.4.1	Methodology	187

7.4.2	Results	188
7.5	Resynchronisation Performance	191
7.5.1	Methodology	192
7.5.2	FTM Resource Cost Measurements.....	192
7.5.3	Network Traffic.....	194
7.6	Probabilistic Conflict Detection/Estimation.....	195
7.7	Contact Manager - A Test Application.....	197
7.8	Further Evaluation	201
CHAPTER 8	CONCLUSION	203
8.1	Contributions.....	205
8.2	Future Work.....	207
8.3	Final Remarks.....	210
BIBLIOGRAPHY.....		210
GLOSSARY OF TERMS		210

Abstract

Disconnected operation based on replication of transaction execution on replicated data items has been demonstrated as an effective technique enabling mobile computers to access shared data in a distributed mobile environment. To guard against inconsistencies resulting from partitioned data sharing (mobile host in disconnected mode), the transaction execution is replicated on a fixed host and upon reconnection the transactions executed on a mobile host and the transactions already validated by fixed host are resynchronised.

This dissertation shows that *twin-transaction model* is a viable solution for a mobile computing environment. It improves data availability and guards against inconsistencies due to partitioned data sharing. The central idea is imposing *one-copy serialisable* requirement on partitioned transaction executions. This requirement is fulfilled by logging/replicating transactions and resynchronising them during reconnection. Transactions executed on a host stay in a tentative state until they pass the consistency validation on server and transit to a resolved state. Invalidated transactions are resolved using provided resolution mechanisms (re-execution, application-specific resolvers, etc.). In addition, the twin-transaction model defines concise conflict representation schemes and enables transaction semantics to be smoothly integrated for conflict resolution.

A working twin-transaction implementation under Windows NT operating system has been developed and used in experiments. The quantitative evaluation based on

controlled experiments and analysis of a sample application establishes that the twin-transaction model incurs acceptable performance and resource overhead.

The main contributions of this thesis are the following: the design of a twin-transaction management system specialised for improving mobile data access and maintaining the consistency; the development of a working twin-transaction system implementation under Windows NT operating system; experimentation and evaluation demonstrating the feasibility and practicality of the twin-transaction system.

The thesis contains 264 pages, 30 figures, 16 tables and 191 references.

Statement

The thesis contains no material, which has been accepted for the award of any other degree or diploma in any university or other institution and to the best of my knowledge, the thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Aamir Rasheed

Acknowledgments

I would like to thank my supervisor, Arkady Zaslavsky, for his unceasing encouragement, support and guidance during the course of this research. Without his valuable advice and assistance this thesis would not have been possible. I benefited greatly from his knowledge and experience on distributed and mobile computing. I would also like to thank my co-supervisor Bala Srinivasan for his valuable help.

My stay at School of Computer Science and Software Engineering, Monash University, would not have been as enjoyable and rewarding without the wonderful people who made it a great place to do research in computer science, especially Maria, Le and Duke.

List of Figures

Figure 2.1 Mobile Computing Environment.....	14
Figure 2.2 Conflict Relation between Read and Write Operations.....	17
Figure 2.3 State-transition diagram for a flat transaction [GR93]	24
Figure 2.4 Business Trip Reservations - graphical representations [WR92]	39
Figure 2.5 Primary Copy method operation [Zas96a]	42
Figure 2.6 Relationship of Mobile Transaction Management to Multidatabase Systems [DHB97].....	66
Figure 2.7 One Possible Kangaroo Transaction [DHB97]	68
Figure 3.1 Mobile Host Execution States	72
Figure 3.2 Twin-Transaction Model.....	77
Figure 3.3 Twin-Transaction Management	79
Figure 3.4 The Twinning Process.....	82
Figure 4.1 Twin-Transaction states in disconnected mode.....	98
Figure 4.2 Twin-Transaction states in connected mode	99
Figure 4.3 Precedence Graph.....	105
Figure 4.4 Architecture of π -Database	124
Figure 5.1 MTM states	130
Figure 5.2 Resynchronisation Algorithm.....	134
Figure 5.3 Data item states and their allowed transitions	137
Figure 5.4 Transaction Resolution Process.....	141
Figure 6.1 Twin-Transaction System Architecture.....	152
Figure 6.2 Main data structures in Transaction Database.....	154
Figure 6.3 Main data structures in Twin-Transaction structure.....	156
Figure 6.4 Item list Node (DI-Node) Structure.....	158
Figure 6.5 Algorithm to resynchronise unmodified data items.	168
Figure 6.6 Algorithm to synchronise transactions in connected mode	169
Figure 6.7 Algorithm to validate a transaction on reconnection.....	170

Figure 7.1 Performance overhead	184
Figure 7.2 Disk space overhead	190
Figure 7.3 Memory overhead	190
Figure 7.4 Disk and memory resource costs	200

List of Tables

Table 6-1 Services for application specific resolvers	173
Table 7-1 TTM Configurations	178
Table 7-2 Benchmark Transactions	180
Table 7-3 Performance overhead ($\epsilon = 10\%$).....	182
Table 7-4 Performance overhead ($\epsilon = 10\%$).....	183
Table 7-5 Performance overhead ($\epsilon = 30\%$).....	184
Table 7-6 Memory overhead	189
Table 7-7 Disk space overhead.....	189
Table 7-8 FTM Resource Cost	192
Table 7-9 FTM Transaction Resolution Costs.....	193
Table 7-10 Network traffic overhead.....	195
Table 7-11 Probabistic conflict detection: memory performance.....	196
Table 7-12 Probabistic conflict detection: disk performance	196
Table 7-13 Probabistic conflict detection: network performance	197
Table 7-14 Probabistic conflict detection: performance	197
Table 7-15 Network overhead and transactions resolved	201

Publications

- [1998] A. Rasheed and A. Zaslavsky. Twin-Transactions – Delayed Transaction Synchronisation Model. In *III Workshop on Mobility and Replication*, Belgium, July 1998.
- [1998] X. D. Zhou, A. Zaslavsky, A. Rasheed and R. Price. Efficient Object-Oriented Query Optimisation in Mobile Computing Environment. In *Australian Computer Journal*, 30(2), May 1998.
- [1997] X. D. Zhou, A. Zaslavsky, A. Rasheed and R. Price. Efficient Object-Oriented Query Optimisation in Mobile Computing Environment. In *Proceedings of the 2nd MCDA Workshop*, pages 76-90, Australia, November 1997.
- [1997] A. Rasheed. Maintaining Data Consistency in a Distributed Mobile File System using Twin-Transactions. In *Proceedings of the 2nd MCDA Workshop*, pages 57-63, Australia, November 1997.
- [1997] A. Rasheed and A. Zaslavsky. A Transaction Model to support Disconnected Operation in a Mobile Computing Environment. In *OOIS' 97, 4th International Conference on Object-Oriented Information Systems*, Australia, November 1997.
- [1996] M. Faiz, A. Rasheed and A. Zaslavsky. Primary Copy Method and its Modifications for Database Replication in Distributed Mobile Computing Environment. In *IEEE Symposium on Reliable Distributed Systems*, Canada, October 1996.

- [1996] A. Rasheed and A. Zaslavsky. Ensuring Database Availability in Dynamically Changing Mobile Computing Environment. In *Proceedings of the 7th Australian Database Conference*, pages 100-108, Australia, January 1996.
- [1996] A. Rasheed and A. Zaslavsky. Transaction Management Models in Distributed Mobile Computing Environment. In *Proceedings of the 1st MCDA Workshop*, pages 18-24, Australia, February 1996.
- [1996] B. Mitelman, S. Lai, M. Nolan, A. Zaslavsky, P. Granville, A. Rasheed and X. D. Zhou. Application Oriented simulation of a Mobile Computing Environment. In *Proceedings of the 1st MCDA Workshop*, pages 106-113, Australia, February 1996.

In the course of the project significant software code was developed that is in excess of 20,000 lines. The software is written using MS Visual C++ under Windows NT operating system.

CRC DSTC funded project on mobile computing will incorporate the twin-transaction model as one the major components.

Chapter 1

Introduction

1.1 Research Background

Database systems are employed to model, store, and retrieve large volumes of data and multiple users manipulate the data at the same time. Database systems are among the most important applications of computer technology. The primary goals of the database systems are to provide multiple users with an organised and uniform method for handling real world data while preserving the integrity and consistency of data. Special application programs (Transactions) are used to interact with databases. A transaction is a sequence of operations, which transforms a database from one consistent state to another consistent state.

Conventional database systems are passive. In other words data is created, deleted, and retrieved only in response to operations issued outside the database system, either by users or by application programs.

Active database systems enhance the functionality of conventional database systems. The database system itself issues operations in response to certain events occurring or certain condition being satisfied. Active capability is captured by the Event-Condition-

Action (ECA) *rule* paradigm. When an *event* is detected, an *action* is triggered, if the *condition* holds. The event-part of an ECA rule specifies what causes the rule to be triggered. Once the rule is triggered, the condition-part is considered. A condition is a database predicate or a database query evaluating the database-state at the time of event detection. If the database predicate is true or the database query produces a non-empty answer, the action in the action-part is executed. An action is an executable program, which may contain data modification or data retrieval operations, transaction operations or arbitrary procedure calls. A brief introduction into the field of active database systems is given in [DGG95], whereas [WC96] contains a broad overview including a discussion of the application areas, integrity constraint management, view maintenance, workflow management and energy management.

A distributed database is a collection of multiple, logically interrelated databases distributed over a computer network [ÖV91a]. A distributed DBMS is then defined as a software system that permits the management of the distributed database and makes the distribution transparent to the users [ÖV91b]. Distributed DBMSs provide transparent access to physically distributed database. The users may issue transactions at any site in the system [Bha86, CP84].

User queries and transactions access the database based on an *external scheme* defined over a *global conceptual schema* which is partitioned and stored at different sites forming the *local conceptual schemas* at those sites. Transactional accesses are managed by the global transaction managers with the help of schedulers for synchronisation and local recovery managers, for reliability.

Traditionally transactions are expected to satisfy the following four conditions. These are known as the ACID properties [Gra81]:

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, ie., if executed alone, the transaction takes the database from one consistent state to another consistent state. When transactions are executed concurrently, the database management system must ensure that the execution of a set of concurrent and correct transactions also maintains the consistency of the database.
3. *Isolation* requires each transaction to observe a consistent database, ie., 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.

To obtain these properties, two main problems should be handled:

1. The effects of several concurrent transactions on the database must be as if they have been executing serially in some order. This property is known as *concurrency control*.
2. The effects of completed transactions must be made permanent and the effects of incomplete ones removed. This property is known as *recovery*.

The two properties make up a *transaction model*.

In a distributed database system a site may simply fail by crashing in a benign manner. Communication links may fail to deliver messages. Combination of such failures may lead to *partitioning failures* [DGS85], where sites in a *partition* may communicate with each other, but no communication can occur between sites in different partitions. It is usually assumed that network provides a routing protocol, which ensures that a message can be delivered between two sites if there is a communication path between them. However, the system is asynchronous; ie, messages may take arbitrary long periods of time to deliver messages.

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 site where the transaction objects are stored.

The notion of transaction, which is the central concept in database and information systems, has evolved over time. Despite their appealing theoretical elegance, the ACID properties have been challenged in different ways and a variety of models are proposed for different environments.

1.2 Mobile Computing

A mobile computing environment is a distributed computing environment, characterised by the fact that some of the hosts (workstations) are mobile while others are stationary (fixed) ones. A mobile host is characterised by a number of connections and disconnections, which in turn make hosts' availability non-deterministic. This results in non-availability of data residing on that particular host. Also mobile host machines are often resource poor relative to their stationary counterparts.

1.2.1 Disconnected Operation and Data Inconsistency

Mobile hosts usually operate in disconnected mode (as they cannot stay connected to fixed network due to poor resources relative to their stationary counterparts). The essence of disconnected operation is to enable a disconnected host act as a temporary server and continue to service the user/application requirements using the (cache or replicated) copies of the requested data. The effects of disconnection cannot always be hidden from the applications running on the mobile host unless the mobile host only accesses the data that is available locally. The disconnected operation on the local copy of the data is achieved by relaxing the traditional partitioned replication control necessary for maintaining one-copy equivalence [DGS85].

The disconnected operation of a mobile host is essentially a special case of a partitioned network. The data made available by disconnected operation may result in conflicting partitioned sharing during disconnected operation. These conflicts can be of two types:

- write/write, and
- read/write

In *write/write* conflict, the data (object) is updated on both a disconnected client and on the corresponding servers.

In *read/write* conflicts, the data (object) is updated in one partition (mobile host or corresponding server) and is read in another.

The process of conflict resolution starts with successful detection of these conflicts. If partitioned sharing conflicts are not detected, then they can cause data inconsistencies in various ways. This is demonstrated by the following examples.

Example 1: John is a field officer and his job is to go to client sites, determine the problems with their equipment and allocate time-slots of different technicians to fix the problem. Back at the office, phone operators are also booking time-slots of technicians for different customers. John uses a disconnected laptop to book these time-slots. Before disconnecting, he cached the time-slots available for a certain technician Arnold and then booked a time-slot of Arnold. In the mean while, one of the telephone operators also booked the same time-slot of Arnold for another customer. The *write/write* conflict on booking of the specified time-slot will leave the database system in an inconsistent state.

If John's laptop remains connected to the servers, either John or the telephone operator will detect the inconsistency at the time of booking and another time-slot could be booked. However, data access/update using disconnected operation resulted in a conflict, which requires aborting one of the operations (booking of the time-slot). A simple abort is not acceptable, the originator of the operation (John or telephone operator) wants to know that such a booking has been aborted, so that they can perform necessary action in this regard (for example, book for another time-slot). They might want to define this procedure as a rule, so that in future similar conflict could be handled/resolved automatically.

Example 2: Jack is a marketing manager and is preparing a report to ask for budgets for marketing those items which are not sold according to expectations. His report is based on the sale figures. He is using a disconnected laptop during a weekend. Before disconnecting he cached a spreadsheet with latest sale figures of each item. While he is preparing his report, the sale figures of some items changed due to late arrival of reports from some sub-offices (the spreadsheet is updated). The *read/write* conflict occurred. The report prepared by Jack is not consistent with the latest figures and during his forthcoming meeting, his report will show old sale figures, while his bosses will have the latest figures.

In this particular example, in the absence of disconnected operation, the *read/write* conflict will still occur due to lack of coordination between users. However, data access using disconnected operation substantially increased the chances of such conflicts and thus inconsistencies. The consistency of the report can be protected, if Jack is notified upon reconnection that the spreadsheet data has been updated. Such notification is also useful, even when disconnected operation is not in use.

1.3 Motivation and Objectives

A mobile computing environment is a special case of distributed computing environment. In a distributed computing environment, hosts are interdependent whereas in a mobile computing environment not only those hosts are interdependent (connected mode) they are also autonomous (disconnected mode). Mobility exacerbates the tension between autonomy and interdependence that is the characteristic of all distributed systems.

In chapter 2, we argue why the existing transaction models cannot proceed when a mobile host is unavailable (non-availability transforms a simple transaction into a long running transaction). Neither making a nested transaction, nor taking savepoints will help, because in either case, the ACID properties are still maintained for the entire lifetime of a transaction. The transaction models that waive some of the ACID properties have no way of insuring data consistency in the face of network partitions (disconnected operation of mobile hosts, a controlled network partition). Additional structuring of transactions does help to organise normal processing, but it does not contain the necessary semantics to cater for long running transactions, which result from non-availability of mobile hosts.

A transaction model targeted at mobile computing environment must be able to handle both (connected and disconnected) modes of operation of a mobile host. A transaction model that allows disconnected mobile hosts to execute transactions must contain mechanisms to achieve a consistent state when the mobile host connects (called *resynchronisation* process). This is to ensure that conflicting transactions are not executed on different hosts when the mobile hosts are disconnected.

A mobile computing system is also a replicated system. Data needs to be replicated; otherwise it will not be available when the host disconnects. In replicated systems, some applications require strong consistency, while others will tolerate some inconsistencies provided that some semantics defined by those applications are kept intact. Therefore it is desirable to design a flexible model that can provide for different consistency modes. It also means that the transaction model should let the application select the required

consistency level and to specify whether notifications of failures and (possibly) successes, under selected consistency mode, is required or not.

1.4 Twin-Transaction Model

Twin-transaction model, presented in this dissertation, is designed to support mobile computing environment. It maintains data consistency during disconnected and connected modes of operations of a mobile host. The central idea consists of four basic elements:

- Logging of transaction histories.
- Optimistic enforcement of serialisability based consistency requirement.
- Automatic detection and resolution of conflicts.
- Probabilistic estimation of conflicts during disconnected mode of operation.

Flexible conflict resolution mechanisms and integration of application-specific semantics are used to help detect and resolve conflicts. The concept of twinning of a transaction is used to incorporate application-specific conflict resolution semantics that are used during resynchronisation of mobile hosts.

It also provides mechanisms to define the semantics of probable conflict detection in disconnected mode of operation based on the learned patterns and/or rules defined by applications. The Twin-Transaction model has the capability to adopt itself to the varying requirements of a computing environment (especially mobile environment).

1.5 Dissertation Outline

Chapter 2 explains the characteristics of a mobile computing environment. The problems that arise under such an environment are also identified. The shortcomings of traditional transaction models under such an environment are discussed. Advanced transaction models and replica control protocols are reviewed and the limitations (under a mobile computing environment) are pointed out. A summary and review of research done to handle transactions, executed by mobile hosts, is also presented.

Chapter 3 defines context of this research, objectives of a transaction model for mobile computing environment and presents the Twin-Transaction model. The *twinning* process and the state transitions of a transaction (the two twins) are explained under the connected and disconnected modes of operation. The two consistency models supported by the twin-transaction model are explained.

Chapter 4 is focused on the execution of transactions in connected and disconnected modes. Transaction execution states and the structure of transaction history log is explained. The proposed concurrency control mechanisms are also presented.

Chapter 5 discusses the resynchronisation process in detail. It is carried out during the reconnection of a mobile host. The resynchronisation process is dependent on transaction history log. The algorithms that are used to perform the resynchronisation process are presented. The data item states during transaction execution and resynchronisation are also explained.

Chapter 6 presents the implementation details of the Twin-Transaction model. The data structures and their usage are discussed in detail. The shortcomings of the implementation are also discussed. The persistence management and crash recovery mechanisms are also explained. The transaction script elements are presented at the end of this chapter.

Chapter 7 analyses and evaluates the performance of the Twin-Transaction model. Different experiments and their results are presented. The resource costs (including network resources) are measured using controlled experiments. A sample application is also used to test the performance and resource cost of the twin-transaction model.

Chapter 6 encompasses the closing remarks and conclusion. The future work and possible extensions to the Twin-Transaction model are discussed.

Chapter 2

Background

In this chapter, the characteristics of a mobile computing environment are presented. A review of classic and advanced transaction models is also presented. Since these transaction models are proposed to tackle different computing environments, an attempt is made to contemplate the suitability of these transaction models to mobile computing environment. Some of the existing replication protocols are also analysed to view their behaviour in a mobile computing environment. A few other techniques to tackle the problems arising under a distributed mobile computing environment are also discussed.

The chapter starts with an introduction to Mobile Computing and discusses the various aspects of mobile computing environment. The traditional and advanced transaction models are discussed in the light of mobile computing environment. Different data replication methods and their applicability to mobile computing environment are discussed next. These include both synchronous and asynchronous replica methods. The related work in the field of mobile computing is discussed last.

2.1 Mobile Computing

Advances in portable computers and rapidly emerging wireless networking technologies provide the basis for mobile computing. Mobile computers are one of the fastest-growing segments of computer industry. Users tend to shift more of their work to

mobile computers (mobile workstations) as they have now the capacity of accessing information resources through wireless connections [BI92c]. The wireless communication technology provides users with the ability to retain their network connection even while moving. At the same time it poses a number of restrictions and problems on a distributed mobile computing environment.

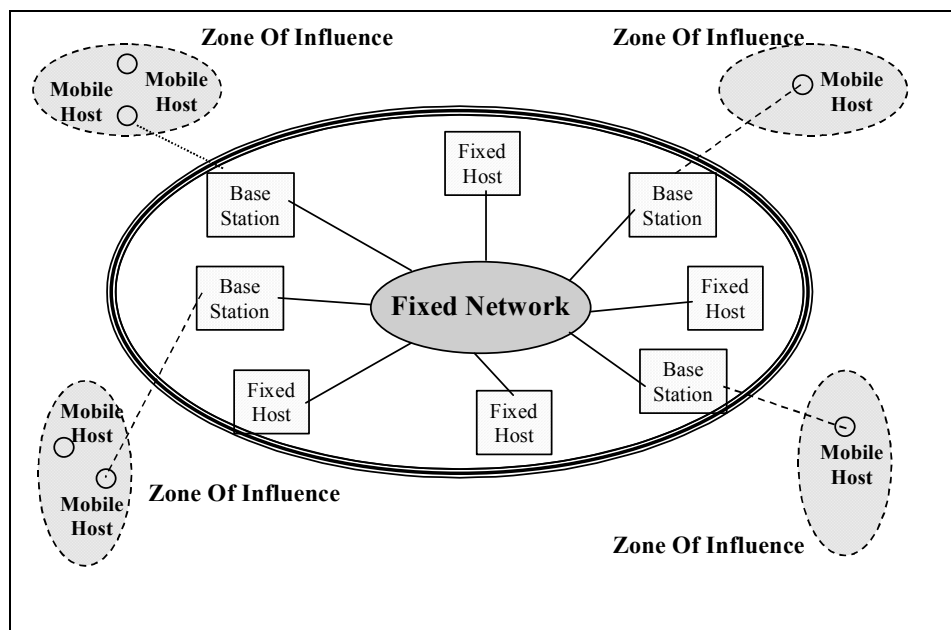


Figure 2.1 Mobile Computing Environment

A mobile computing environment is a distributed computing environment, characterised by the fact that some of the hosts (workstations) are mobile while others are stationary (fixed) ones. Some of the fixed hosts, called *base* nodes are augmented with a wireless interface to communicate with mobile hosts. The area covered and controlled by a base node is called a zone of influence (or a logical *cell*). This area can either be a territorial division or a logical cluster of mobile and/or fixed workstations. At any moment, a mobile host can directly communicate with only one base node, regardless of whether it is a home base node, or a visitor base node. A hypothetical architecture of a mobile computing system is depicted in Figure 2.1.

The mobility of hosts has an impact on many system level services like database management systems (DBMS) and distributed operating systems. Changes and enhancements to operating systems, as well as other system level additions are discussed in [BI94, AK93, IK95 and IK96].

When a mobile host disconnects from the rest of the network (fixed network) it operates in a disconnected mode (called a disconnected host). The major issue is the management of disconnected mode of operation (disconnection management). The management of disconnected mode of operation involves streamlining the data access (read/write) operations of a mobile host and on reconnection achieving a state consistent with the rest of the system (hosts). The disconnection management forces a rethink of file system design, concurrency, query processing, data replication and transaction models [PB93, PB94, PS98].

Many portable computers are powerful enough to operate as clients and access shared data. Some of them might as well be able to keep shared data for other clients. Unfortunately, mobile access to shared data (or access to shared data on a mobile host) is constrained in important ways. A mobile host is characterised by a number of connections and disconnections, which in turn make that hosts' availability non-deterministic. Also mobile host machines are often resource poor relative to their stationary counterparts. This will result in non-availability of data residing on that particular host. Thus mobile elements of shared data will not always be available to rest of the network and the data on the network is not always available to mobile hosts.

Mobile hosts usually operate in disconnected mode (as they cannot stay connected to fixed network due to poor resources relative to their stationary counterparts). The essence of disconnected operation is to enable a disconnected host act as a temporary server and continue to service the user/application requirements using the (cached or replicated) copies of the requested data. The effects of disconnection cannot always be hidden from the applications running on the mobile host unless the mobile host only accesses the data that is available locally. The disconnected operation on the copy of the data is achieved by relaxing the traditional partitioned replication control necessary for maintaining one-copy equivalence [DGS85].

Support for disconnected operations is very important for mobile computing systems as the moving host can frequently disconnect from the network (when dropping out of the coverage areas, trying to save battery life). Unlike traditional distributed systems, this should not be treated as a failure. Implicit support via the operating system, or explicit support embedded in the application should enable continued processing.

Replication is necessary in a distributed mobile computing environment because of the very nature of a mobile host. Mobile hosts cannot remain connected to the rest of the system all the time. This is mainly due to two reasons; firstly, to save resources on a mobile hosts and secondly, a mobile host can go out of a zone-of-influence, thus causing disconnection. Replication in mobile database systems is discussed in [FZ95, Zas96a, Zas96b, ZZ97] and is discussed in detail in section 2.4.

The disconnected operation of a mobile host is essentially a special case of a partitioned network. The data made available by disconnected operation may result in conflicting

partitioned sharing during disconnected operation. In a transaction system, two operations conflict if both of them are on the same object and at least one of them is a write operation. Figure 2.2 illustrates the conflict relation with respect to a single object. A *No* entry in the table indicates that operations do not conflict where as a *Yes* entry dictates that operations do conflict. The conflicts depicted in Figure 2.2 are of two types and both of these conflicts can occur in a mobile computing environment.

	Read	Write	
Read	No	Yes	Read
Write	Yes	Yes	Write

Figure 2.2 Conflict Relation between Read and Write Operations

In *write/write* conflict, the data (object) is updated both by the disconnected client and by the corresponding servers. In *read/write* conflict, the data (object) is updated in one partition (mobile host or corresponding server) and is read by the another.

The process of conflict resolution starts with successful detection of these conflicts. If partitioned sharing conflicts are not detected, then they can cause data inconsistencies in various ways.

In section 2.2, traditional transaction models and their inadequacy for mobile computing environment is discussed. Then in section 2.3 suitability of some of the major advanced transaction models is discussed in light of mobile computing environment. Replication is one of the promising mechanisms to overcome the non-availability of data in a mobile computing environment. A brief survey of replication protocols is presented in section 2.4. Different mechanisms that are presented in literature to tackle the problems of a distributed mobile computing environment are discussed and their shortcomings are outlined.

2.2 Traditional Database Systems

A database is a collection of *data items*, which satisfy a set of integrity constraints [Elm92]. The state of the database is defined by the *values* of these data items at any one time. The size of the data contained in a data item is called the *granularity* of the data item.

A transaction accesses and manipulates the data items in the database by invoking operations specific to individual data items. A transaction that updates the database must preserve the integrity constraints of the database. A transaction is defined in terms of the following important notions.

Visibility:

Referring to the ability of one transaction to see the results of another transaction while it is executing;

Consistency:

Referring to the correctness of the state of the database that a committed transaction produces;

Recovery:

Referring to the ability, in the event of failure, to take the database to some state that is considered correct; and

Permanence:

Referring to the ability of a transaction to record its results in the database.

Traditionally, transactions are expected to satisfy the following four conditions, known as ACIDity properties [Gra81]:

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, ie., if executed alone, the transaction takes the database from one consistent state to another consistent state. When transactions are executed concurrently, the database management system must ensure that the execution of a set of concurrent and correct transactions also maintains the consistency of the database.
3. Isolation requires each transaction to observe a consistent database, ie, 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.

Transactions and their ACID properties (atomicity, consistency, isolation, and durability) have traditionally been used to ensure consistent database management and atomic and isolated user actions. One of the major goals in such systems is to allow several users to access the information simultaneously while the integrity of the system is preserved. Assuming that each transaction executing alone transfers the system from one consistent state to another consistent state, two main problems should be handled:

1. Controlling the interactions of concurrent transactions so that they do not corrupt effects of each other while still executing and this problem is known as concurrency control.
2. Making sure that the effects of completed transactions are made permanent and the effects of incomplete ones are removed. This problem is known as recovery.

2.2.1 Concurrency Control

Concurrency control is the activity of coordinating concurrent access to a database in a multi-user database management system. Concurrency control gives the illusion that each user is executing alone on a dedicated system.

The concurrency control problem is exacerbated in a distributed DBMS because data is distributed among many systems and a concurrency control mechanism on one computer cannot know (at least instantaneously) about interactions at other computers [Pap86]. Thus a global concurrency control mechanisms is required. More than 20 concurrency control algorithms have been proposed for DDBMSs. These algorithms are complex and hard to understand. These algorithms make different assumptions about the underlying DDBMS environment and thus it is difficult to compare them [BG81].

The main approaches to concurrency control are locking, timestamp ordering, and optimistic non-locking mechanisms. The last two are basically proposed for avoiding well-known disadvantages of locking mechanisms [BK91]. One important drawback of locking mechanisms, particularly in distributed environments, is deadlock that happens when two processes wait for each other to release resources. Deadlocks should either be prevented, eg, suspend transactions until they reserve all the resources they need, or detected and broken.

Optimistic non-locking mechanisms basically try to avoid unnecessary locking activities. They assume that the transactions are non-conflicting, almost always, and suspend conflict checking to the last steps when a transaction is ending. If no conflicts arise the transaction is free to commit, otherwise it will be aborted.

Timestamp ordering guarantees that a deadlock situation cannot arise. A *data manager* orders conflicting access to data items by multiple transactions according to their timestamps. A protocol will decide which one can go, must wait, or must abort (and restart later). Timestamps provide a *total ordering* of transactions to be used for controlling concurrent access to the same data item.

Although timestamp ordering solves the above problems by not using locks and not allowing conflicting access to data, its main drawback is restricting concurrency. A transaction may be done with a data item, but while still being busy somewhere else, it may cause others to unnecessarily wait or abort. This can be improved by using transaction semantics that allow others to access the data item concurrently if no inconsistency can arise.

Semantic-based synchronisation can be broadly divided into two groups. Models of the *data approach* define concurrency properties on abstract data types according to the semantics of the *type* and its methods. Semantic knowledge about individual types is used to develop synchronisation strategies that allow more concurrency. An object offers a concurrent behaviour regardless of the semantics of applications using it. The interleaving of concurrent transactions is *implicitly* constrained by operation conflicts defined on abstract data types. Models of the *transaction approach* define concurrency properties on transactions according to their semantics and the data they manipulate. Here, interleaving is *explicitly* constrained by specifications on transactions. This approach requires centralised control with respect to concurrent transactions.

2.2.2 Recovery

Recovery basically deals with failures that may prevent in-progress transactions from completion or remove effects of ones already completed. Both cause inconsistencies due to violation of *atomicity* and *durability* properties of transactions. A good system should be able to recover from most type of failures without human intervention.

Failures are of different types. Ones relating to data entry or transaction programming errors are beyond the scope of this thesis. Other types of failures irrelevant to this discussion are operators' errors and hardware errors. Failures, that are directly relevant to transaction management [BHG87] include transaction failure, system failure and media failure. There are additional types of failures related to distributed-transaction management. *Site failures* refer to *partial* or *total* failure of sites in the network, ie, some or all of the sites being down. *Communication failures* refer to failure of communication links between sites. Some or all communication paths between two sites

may fail. A *partition* situation happens when two or more site clusters can only communicate internally. A popular mechanism for detecting failures of other sites is by *time-out*. An extra requirement for distributed transaction management is global commit synchronisation. Popular methods are two-phase and three-phase commit protocols [BHG87].

The recovery manager in a database management system takes care of committing, aborting and restarting transactions. It ensures that the stable storage has the data needed to restart a failed transaction. Updates may be done *in-place*, which destroys the old copy each time a data item is overwritten (keeping one copy at a time), or by *shadowing* which keeps older versions as shadow copies. The recovery manager usually stores additional information in a stable storage to *log* the history of execution. This information may be needed for *undoing* the effects of uncommitted transactions and *redoing* the effects of committed ones if failure occurs.

Efficiency of the recovery manager is very important since doing a restart prevents all users from accessing the database. Therefore the size of the *log* is a crucial factor. The problem is solved by *check pointing*, which is an activity that writes information to stable storage during normal operation in order to reduce the amount of work the restart procedure has to do after a failure. The restart procedure fails if *media failure* destroys the needed information. The only recourse is to maintain redundant copies of every data item's last committed value. Keeping more copies (at different places, etc.) increases the probability of recovery.

2.2.3 Impact of Mobile Computing Environment

A mobile computing environment is characterised by a number of connections and disconnections, which in turn make mobile workstations' availability non-deterministic. When a mobile workstation (MWS) is participating in a distributed transaction that involves more than one host, then due to potential non-availability of a mobile workstation, a relatively short transaction can become a long running transaction waiting for the mobile workstation to respond. A long running transaction will then hold the locks and will block the system resources.

In the following paragraphs, traditional transaction models and some of the advanced transaction models are discussed in light of availability of the mobile hosts.

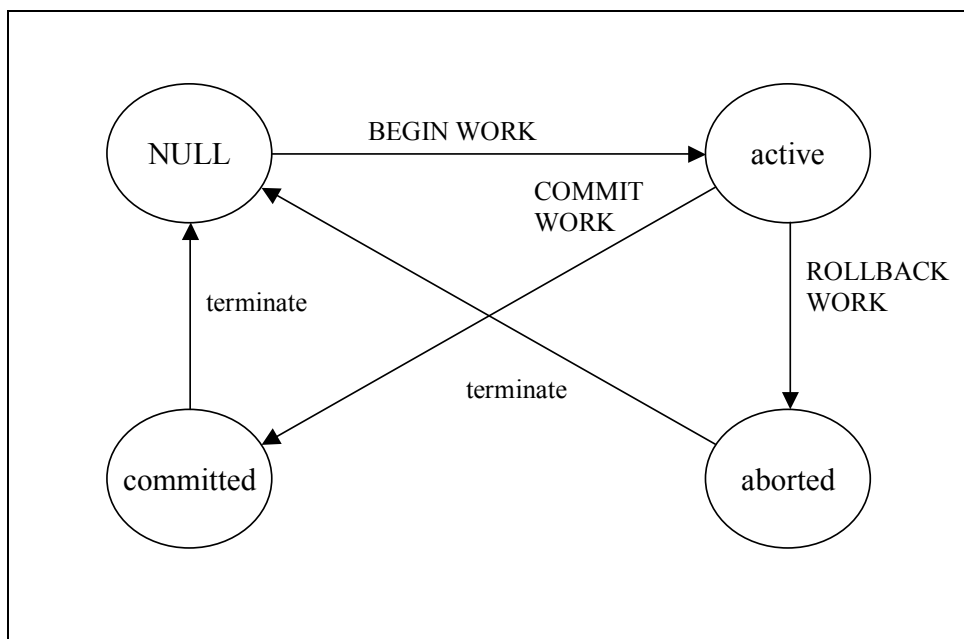


Figure 2.3 State-transition diagram for a flat transaction [GR93]

Flat transactions [Esw76] are the simplest kind of transaction management models and represent the most popular and common type of transaction. These transactions are called flat because there is only one layer of control. That is, the transaction will either

survive (commit), or it will be rolled back (aborted) as shown in Figure 2.3. In flat transactions there is no way of either committing or aborting parts of a transaction, or committing results in several steps. Flat transactions give the simplest failure semantics, but if anything goes wrong, there is only one option, rollback and start the transaction from the beginning. An early extension to the flat transaction concept was to have savepoints [Ast79], which can serve as restart points in case of a failure.

In case of flat transactions, if an MWS is participating in a transaction, then the mobile host must be available for the whole period of a transaction (to participate in the commit protocol at the end of a transaction). One possible solution is to allow the MWS to disconnect from the network after finishing its part of a job and when required, a connection can be re-established to inform about the commitment or abortion of a transaction. A savepoint can be inserted whenever a portion of transaction finishes on the mobile workstation, so that the MWS would not be contacted to do the same work again.

Chained transactions [BHM90] are a variation of flat transactions with savepoints. The idea behind chained transactions is that rather than taking savepoints, the transaction is divided into multiple commit-points. After a commit, the transaction cannot be rolled back. The advantage of this approach is that in case of a failure, the transaction can start from the last commit-point. Also, any locks that are no longer required can be released at each commit-point. The intermediate results produced after each commit-point are not visible until the whole transaction commits. In case of an abort, only the last uncommitted portion of the transaction can be rolled back, the application generating the transaction has to handle what to do in such a scenario.

Chained transaction is a set of subtransactions where commit of one subtransaction marks the start of another subtransaction and the commit of last subtransaction of the set also commits the whole transaction.

Chained transactions will perform better as compared to flat transactions in a mobile environment when processing required on a mobile host is submitted as a subtransaction. In case of flat transactions (with savepoints), even when a savepoint is inserted after completion of processing on an MWS, the transaction can still fail as a whole and rollback has to be performed on the MWS as well, and possibly the entire transaction will run again. In chained transactions, if a commit was inserted after the completion of processing on an MWS, it will only be contacted when the transaction finishes (commits or aborts).

Nested transactions [Mos85] are a generalisation of savepoints. Whereas savepoints allow organising a transaction into a sequence of actions that can be rolled back individually, nested transactions form a hierarchy of pieces of work. A nested transaction is a tree of transactions (subtransactions), each of which is either a nested or a flat transaction. A subtransaction can commit or rollback, but its commit will not take effect, unless the parent transaction commits. The commit of a subtransaction makes its results accessible only to the parent transaction. If parent transaction rolls back, all its subtransactions are also rolled back. The nested transactions give better control and structure as compared to flat transactions (with savepoints). They can be used to increase the level of parallelism in executing a transaction.

Subtransactions in a nested transaction are not fully equivalent to classical flat transactions. They are valid only within the confines of the surrounding higher-level transaction. They are *atomic* from the perspective of that parent transaction; they are *consistency preserving* with respect to the local function they implement; they are isolated from all other activities; but they are not durable because even though they commit, they will be rolled back if their parent transaction rolls back.

Since a nested transaction is not committed until the root (top-level) transaction commits, the work done on an MWS might be required to be redone in case of an abort or failure. Chained transactions will perform better than nested transactions because an early commit (sub-transaction commit) is allowed.

2.3 Advanced Transaction Models

Traditional database systems are found lacking in *functionality* and *performance* when used for applications that involve reactive (endless), open-ended (long-lived) and collaborative (interactive) activities. Hence, various extensions to the traditional model have been proposed [Elm92]. Compared to the traditional transaction model, these models associate *broader* interpretations with the four transaction properties mentioned in section 2.2. These broader interpretations help provide enhanced functionality while increasing the potential for improved performance.

Atomic actions are the basic building blocks for structuring a model. Atomicity requires precise specifications for two fundamental aspects of each operation. One is the question of when and how the results of the operation are made accessible to other operations in the system; this has to do with completing an operation successfully. The

other aspect is how the partial results of an operation that does not complete successfully be rolled back (or compensated) such that no side effects (or only well controlled side effects) occur.

The ACID properties of conventional transaction models are not appreciated in advanced applications, particularly ones, which involve long-duration transactions. In [MP93], the authors have presented five examples of long running transactions and why traditional transaction models do not apply to those particular transaction examples.

In literature, a number of extended transaction models are proposed. Most of these transaction models deal with improving concurrency control and to support long-lived transactions. In these transaction models, the non-availability of a data item is not discussed. We summarise the most popular ones in this section. A weakness of many of the following transaction models is that the disconnected operation of transactions and delayed conflict-detection and resolution was not envisaged as one of the requirements of a transaction model. This requirement emerged due to the very nature of a mobile computing environment.

2.3.1 Sagas

Sagas transaction model [MS87] was proposed as a transaction model for long-lived activities. A *saga* is a set of long-duration transactions comprising a set of (component) subtransactions that can be interleaved in any order with subtransactions of other sagas. Component transactions within a saga execute in a predefined order (eg, sequential or parallel).

Each subtransaction is associated with a compensating subtransaction. A saga requires that either all component subtransactions complete execution or *compensating* transactions are ran to undo the effects of ones that have committed before failure of the saga. This does not necessarily mean that the database is restored to the state that existed when the transaction began. Both component and compensating transactions behave like atomic transactions (traditional mode) and preserve the ACID properties but their behaviour is constrained by certain dependencies (eg a compensating transaction may only execute after failure of its counterpart).

A saga commits when all of its components commit in the prescribed order. A saga is not failure atomic, but it cannot execute partially. Thus when a saga aborts, it has to compensate for the committed components by executing their corresponding compensating transactions.

Sagas may view the partial results of other sagas since component subtransactions commit independently. Therefore, consistency in sagas is not based on serialisability. Failure of a component forces the whole saga to abort. In this respect, sagas do not have the flexibility that most of the advanced transaction models have, ie, they are not able to retry an aborted component or ignore it or try an alternative instead.

When a saga is running in a mobile computing environment, some component, say C of saga might involve mobile hosts. Since components need to be executed in a prescribed order, all the components that are to be run after C (sequential order) will have to wait until the mobile host becomes available and that particular component finishes. If at a later stage the saga is to be aborted, the compensating transaction has to wait again for

that mobile host (if its not available). Since the mobile host might not become available for a long time, the execution of saga will be interrupted either while executing or aborting and might requite user intervention to restore system-state.

2.3.2 Multilevel Transactions

The main features of the multilevel transaction model (also known as layered transactions) [Wei91] [WS92] are exploiting the semantics of operations to relax isolation of concurrent transactions, using compensating transactions to achieve atomicity, and making partial results of subtransactions visible to other concurrent transactions.

The model provides parallelism and supports long-duration transactions by utilising the semantics of operations in an object-oriented manner. It is a special case of open nested transactions [WS92] in which nodes of transaction tree correspond to execution of operations at particular levels of abstraction in a layered system. Hence all transaction trees have the same height which is equal to the number of levels in the underlying system architecture.

It uses a multilevel concurrency control mechanism in which the *semantics of level-specific operation* is exploited to handle conflicts. Conflict relations are defined on operations as specified at particular level of abstraction rather than operation execution. The high-level operations are implemented by read and write access to the underlying records. For example, if accounts a and b belong to the same branch and *deposit* operations update a branch total as well, their concurrent access to it is regarded as *pseudo conflict* and the schedule is regarded as serialisable at the top level.

If, in a two-level system, the conflict relation at the higher level is empty, the model is similar to *sagas*, which is based on the assumption that all high level steps are conflict free. A transaction is decomposed into a sequence of independent subtransactions. A concurrency control criterion called multilevel serialisability is developed which is based on the assumptions that all recovery-related steps are explicit actions in the transaction schedule and are subject to concurrency control, and resulting complete schedule is multilevel serialisable in the ordinary sense.

The multilevel transaction model is relatively conservative compared to other models, but is fairly powerful and is shown to be applicable in a number of applications including extensible DBs, federated DBs, operating system transactions, and object oriented DBs [WS92].

Multi-level transactions have the benefits of nested and chained transactions. Persistent subtransactions relax the atomicity of their parents since they survive the abort of a parent. It may be reasonable in a mobile environment to specify the subtransaction executed on a mobile host as persistent. The operations, that are to be run on a mobile host, can be made non-compensatable and these operations can be executed when the transaction commits. The execution of these operations can be deferred till the mobile host becomes available.

2.3.3 Split Transactions

Split Transactions [PKH88] are used for open-ended applications, such as VLSI design, CAD/CAM and software development activities. Open-ended activities are characterised by *uncertain duration* - from hours to months; *Unpredictable*

developments - actions unforeseeable at the beginning; and *Interaction with other concurrent activities* - for concerted efforts among users.

To solve these problems, dynamic restructuring of on-going transactions is proposed. This restructuring is done by two special operations namely the *split-transaction* and *join-transaction* operations. The purpose is to split the objects of an on-going transaction among two or more serialisable transactions. Certain concurrency properties are defined for operations of resulting subtransactions to ensure consistency.

Split transactions are useful for committing part of a transaction early, thereby releasing resources as well as useful results of the original transaction. The model reduces isolation property and saves parts of work from subsequent failures. Furthermore, defining the inverse operation of split, i.e. join-transaction, eases serialisable access to resources. On-going serialisable transactions are joined as if they had been a single one. Hence a split can achieve transfer of resources where a resulting transaction joins another on-going one. The combination is useful in long-duration transactions. The real application is what the authors call user-controlled transactions where the operations are selected by the user as they go along. The transaction manager provides certain user commands for this purpose.

The isolation property is relaxed and parts of work from subsequent failures are saved. Split transactions under mobile computing environment will result in less amount of work to be undone, but it only solves the problems for open-ended applications (CASE, CAD, CAM, GIS environments).

2.3.4 S Transaction

The S Transaction (Semantic Transaction) model [EVT88] is a variation of nested transactions with some flexibility. Due to the importance of local autonomy, the issuer of the top-level S Transaction does not dictate which further systems should participate. If a participant refuses to process a subtransaction or if a failure arises, an alternative system could be tried. Furthermore, compensating transactions are used for recovery. Hence the isolation property is relaxed to subtransaction level.

Traditional atomicity is replaced by a semantic one. A top level S Transaction either does all what must be done or cancels incomplete attempts in such a way that no *semantically* effective results are left in any database. If automatic recovery reaches its limits, a human intervention will rectify the situation. The probability of such a semantic crash is decreased by trying to run local compensating programs several times. S Transactions are dynamically generated.

No particular concurrency control or commitment protocol is recommended at the top level. The local transactions are the concurrency units. Five types of autonomy, namely organisational, design, management, communication and execution autonomous are addressed in the model. A component database is even free to break a communication process in the middle, or may refuse to execute a commit or abort message in a global synchronisation protocol. A timeout mechanism is used to control late subtransactions.

The model was developed for an inter-organisational autonomous banking system. It is however more general and could be used for other application domains where local

autonomy is the main goal. In a mobile computing environment, if the local autonomy of each host is to be preserved then S Transactions will fit well.

2.3.5 Flex Transactions

The work on flexible and multidatabase transactions [ELLR90], [RELL90], [RKC92] is based on the observation that failures of individual transactions may be tolerated as a transaction may be accomplished by more than one local database system. They provide a framework where the designer may specify atomicity requirements of subtransactions as well as their precedence and data flow requirements.

Unlike sagas, the model supports the concept of mixed transactions allowing compensatable and non-compensatable transactions to coexist within a single global transaction. It also incorporates the concept of time in scheduling of transactions and subtransactions. A global transaction in this model is syntactically a two-level nested transaction, but its semantics are expanded by allowing function replication, independent commitment of some subtransactions before the corresponding global transaction, and the specification of the value of completion time of (sub)transactions. Users are allowed to specify alternative subtransactions or sources of data for implementing the same task. The temporal dependency of the model supports transaction execution and completion orderings.

The concepts of positive and negative dependencies are introduced for execution ordering and defining alternative subtransactions respectively. To facilitate the execution dependency specification, a *transaction execution state* is defined as an n-tuple which specifies if a subtransaction has been submitted, successfully completed,

unsuccessfully completed, failed, or is being executed. Also for each subtransaction an acceptable state set as well as a precedence predicate is defined. A global transaction is also defined as a tuple that specifies all its aspects. Transactions are scheduled using the execution rules that are defined. The Predicate Petri Nets [LM86] are used to control the execution of global transactions.

In flex transaction model, since users can specify alternative resources, the transaction will know what to do if a certain MWS is not available. This puts some extra burden on the user issuing the transaction, to know more about the semantics of a data item and its alternative sources. The temporal property of a (sub)transaction can be used to specify to execute a transaction at a time when it is more likely that MWS will be available (ie., some heuristic to be built into the database manager).

2.3.6 DOM Transactions

Distributed Object Model (DOM) transaction management [BM92] is intended to facilitate the development of non-traditional applications in a distributed object oriented environment which supports the co-existence of autonomous, heterogenous systems, some of which may be non-database systems (eg. file systems). The model allows a combination of closed nested transactions (called top-transactions) and open nested transactions which relaxes top-level atomicity constraint (called multi-transactions) as well as compensating and contingency transactions and vital or non-vital subtransactions. Rules defining compensating transactions are attached to objects. Dependencies may be specified to force subtransactions to execute or commit in a specific order.

The main contribution of the model is to support disparate requirements such as active capability, heterogeneity, local autonomy, abstract operations and long-duration activities within a single integrated model. It separates the transaction model from correctness criterion. The Transaction model determines capabilities and restrictions for users to write transactions while the correctness criterion determines acceptable concurrent transaction histories. Such a separation allows many transaction management schemes, some of which have been studied by the authors.

Transactions that are specified as component transactions of another transaction may either be *vital* or *non-vital*. A vital transaction must be executed successfully. A non-vital transaction may abort without preventing the parent transaction from committing.

This notion of vital and non-vital transactions can be used in a mobile computing environment. If the transaction component for the mobile host can be declared non-vital, then the transaction can proceed whether or not the mobile host is available. The semantics of applications will dictate whether this is possible for that particular application or not.

DOM transaction model also has *contingency transactions*. A contingency transaction is invoked upon the occurrence of some failure condition and before commit of the transaction for which it is an alternative. When a mobile host is not available, these contingency transactions can be used to create a log of component transactions that are to be ran on a particular mobile host when it becomes available (assuming that these component transactions can be executed successfully when the mobile host becomes available).

2.3.7 Cooperative Transaction Model

The *Cooperative Transaction Model* was proposed for design applications [NZ84]. Design applications require shared persistent data and have different transaction management requirements. Serialisability in the traditional transaction models restricts cooperation between transactions because conflicting operations are executed in a strictly sequential way. To overcome this problem, a tree structure called the cooperative transaction hierarchy is used to reflect the underlying design task. Each of the internal nodes is called a *transaction group*, and they accomplish their tasks through cooperation of their members. The leaf nodes are cooperative transactions. A programmable correctness criterion is defined for controlled data sharing. A set of active patterns and conflicts are used to define the correctness of each transaction group. Conflicts specify when certain operations cannot occur and patterns specify which operation sequences are allowed in a history. A synchronisation algorithm is used to test the correctness criteria for each transaction group.

A user-defined correctness criterion is implemented in place of the traditional correctness criteria defined by serialisability. The cooperation between transactions is allowed by removing *Isolation* property of transactions.

Specific parts of a transaction can be aborted without causing the whole transaction to abort. The effects of these aborted operations are purged from the database.

The ability to define conflicts and patterns gives some ability to define the mobility of a host and thus redefine the correctness criteria when a mobile host changes its state (becomes non-available or available).

2.3.8 ConTract Model

The *ConTract Model* [Reu89, WR92] was proposed to handle long-lived computations in advanced applications, especially in workflow areas. A contract is defined as a set of predefined actions (steps) with control of flow explicitly defined for those steps. The ConTract model must be *forward recoverable*. The execution of a ConTract must be re-instantiated and continued from where it was interrupted due to a failure. To achieve this, the state of the system (each step and global state of the ConTract) must be made recoverable.

In place of conventional concurrency control mechanism, a predicate-based concurrency control mechanism is used (similar to the one presented in [KS94]). The main characteristics of this mechanism are:

- Application dependent degree of isolation by the definition of predicates.
- Support of compensation instead of *rollback*.
- Distinction between predicates to ensure correctness and predicates supporting the successful (forward) execution.
- Reliable (recoverable) concurrency control information outside of transactions.

A new correctness criteria is also introduced which is based on the following two conditions:

- An execution is correct if a final state is reached, or
- It is insured that for every successfully finished activity, the compensating activity can be executed.

Partial results can be made available even before the completion of whole ConTract. Compensating transactions are used to obliterate the results of committed steps that are not needed. The ConTract allows one to resolve conflicts in a more flexible way by specifying what to do when conflicts occur.

APRICOT [Sch93] is a prototype implementation of the concepts of the ConTract model. It is a workflow-programming environment. The ConTract model models control flow using scripts and steps. *Script* describes the control flow and other execution strategies of a long-lived activity. Control flow between related steps is modelled by the usual elements: sequence, branch, loop and some parallel constructs. *Steps* are the elementary units of work in the ConTract model. Each step implements one basic computation of an application (Figure 2.4).

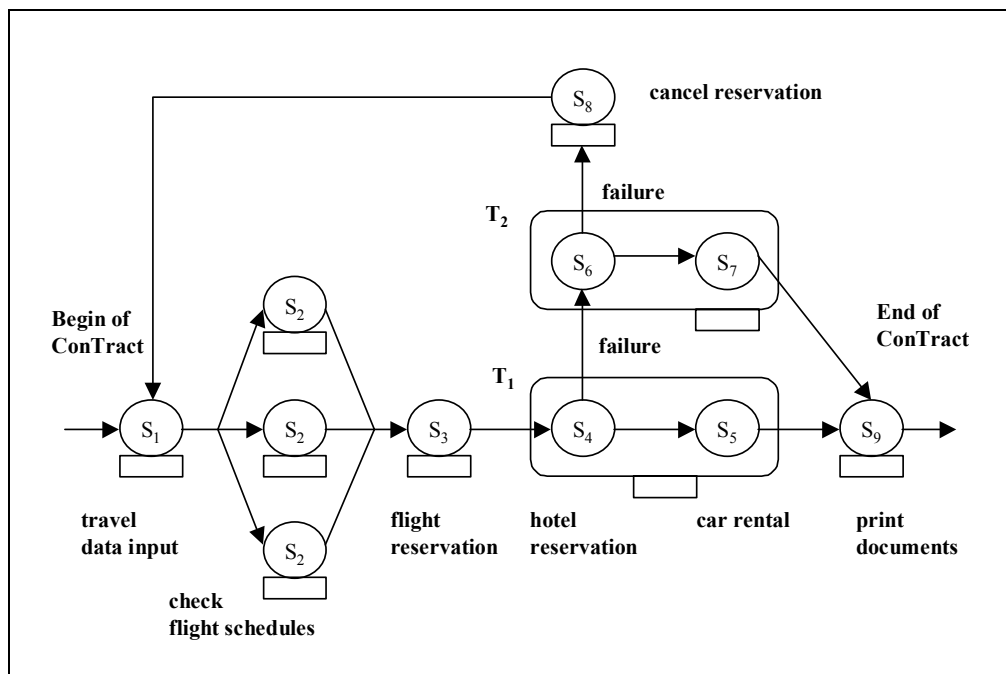


Figure 2.4 Business Trip Reservations - graphical representations [WR92]

2.4 Replication

Distributed databases utilise replication as a means of increasing database availability and reliability. Replication means that some data objects are intentionally stored redundantly at multiple sites. Replication of data objects improves fault tolerance and helps in sharing load between nodes that hold copies (replica). The apparent disadvantage of data replication is concerned with the overheads needed to maintain data consistency across multiple sites.

To execute a transaction on a replicated data, the transaction manager must first translate every operation on a data item into an operation on one (in the case of a Read) or several (in the case of a Write) operations on copies of that data item. Various replica control protocols are devised to furnish read/write requests and propagate changes to other replica [FZ95].

Replication of data items in a mobile environment solves the problem of availability of a data item, but the problem of maintaining the consistency among different replicas of a data item is aggravated. In mobile computing, a host maintaining a replica of a data item can be mobile. The availability of the replica on the mobile host will be dependent on whether or not the mobile host is connected or disconnected. Similarly, the data maintained by the fixed hosts is not available to a mobile host in disconnected mode.

In this section some of the existing replication protocols are reviewed. These replication protocols can be classified as synchronous or asynchronous. The two classes of replication protocols are discussed separately in sections 2.4.1 and 2.4.2. Their

applicability to mobile computing environment is also discussed. A replication method for mobile computing environment should:

- Allow updates in partitioned networks.
- Maintain consistency of data in the face of partitioned data updates.
- Keep communication cost between mobile and fixed hosts to minimum.

2.4.1 Synchronous Replication Protocols

Primary Copy and Quorum Consensus are the two basic replication protocols for managing replicated data synchronously. These two are discussed first, then a number of variations are presented. Surveys of synchronous protocols can be found in [CH91, Son88].

2.4.1.1 Primary Copy

In *Primary Copy* method [AAC92], one replica is designated as the primary copy, all other replicas are secondaries. Only primary copy is allowed to handle *write requests*. Any replica (primary or secondary) can handle a *read* request. Update requests are sent to the primary copy, which acquire locks on all secondaries, performs the update, broadcasts the change to all secondaries and then releases the locks (Figure 2.5).

In the face of network partitions, the partition containing the primary copy is allowed to perform updates. Thus *Primary Copy* method maintains consistency in the face of network partitions. But it does not tolerate the failure of the primary copy. Also, since all the updates are performed on the *primary copy*, the site holding the primary copy can become a bottleneck.

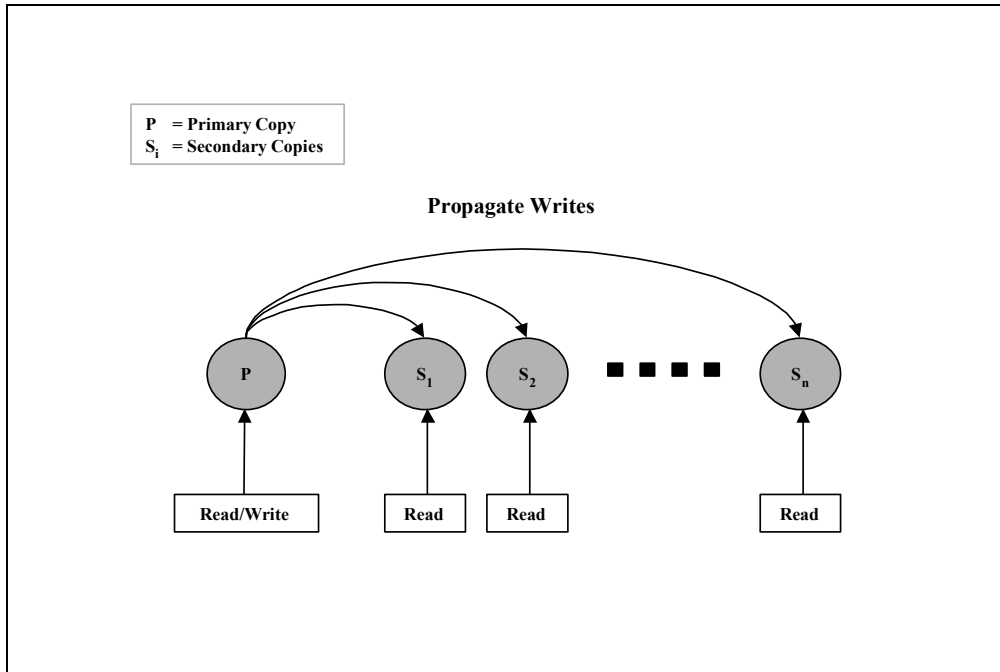


Figure 2.5 Primary Copy method operation [Zas96a]

2.4.1.2 Virtual Primary Copy

In [FZ95, Fai95], it is argued that the traditional replica methods are not suitable for mobile databases in their present state, and the authors have presented a variation of *primary copy* method, namely, *virtual primary copy* (VPC) method. In *primary copy* method [AAC92], one node is designated as the primary copy node. When a data item X is written, a *write* request is sent to the node that holds the primary copy X_p . The changes are propagated to other replicas asynchronously. A *read* request can be sent to the primary copy but accessing the local copy can perform it more efficiently. The *primary copy* method assumes that most of the *write* requests are generated from the host holding the primary copy.

In *virtual primary copy* method [FZ95], if the primary copy X_p of a data item is located on a mobile host, then a copy of that data item X_{vpc} (called VPC) is also maintained on the base node of that mobile host. When the mobile is connected, all the transactions are

performed on the mobile primary copy X_p and the virtual primary copy X_{vpc} is treated as just another replica. In case of a disconnection, the transactions are performed on the virtual primary copy X_{vpc} (which acts as a primary copy). Whenever the MWS connects again, the primary copy on MWS X_p and the virtual primary copy X_{vpc} are synchronised. If there are any conflicts (inconsistencies), they are resolved. This might result in rolling back transactions on mobile hosts' primary copy X_p or virtual primary copy X_{vpc} . Essentially, for all the transactions X_p and X_{vpc} act as one copy.

In VPC, the problem of consistency management among replicas of a data item is resolved. The VPC method decides on a transaction-by-transaction basis whether to execute that transaction on mobile hosts' primary copy X_p or virtual primary copy X_{vpc} . This necessarily requires a transaction to be restarted when mobile host disconnects. Also, when a mobile host connects, it either has to wait for the completion of all transactions (operating on X_{vpc}) before re-synchronising itself with rest of the system or all running transactions will have to be restarted. If the transaction execution process is also replicated then these problems can be resolved.

2.4.1.3 Quorum Consensus (QC)

The second class of replica management protocols is known as *Quorum Consensus* and is designed for distributed systems. To execute an operation, the consensus of a group of nodes is required. This group is called *quorum group*. An operation proceeds to completion only if it can obtain permission from nodes that constitute a *quorum group*.

The groups are determined by the system requirements. Quorum groups used by conflicting operations have non-empty intersection to guarantee proper synchronisation. A majority consensus method for defining a quorum set is presented in [Tho79]. Gifford

[Gif79] presented a generalisation of this method. Consider n replicas of an object. Each replica is assigned some number of votes that can be used in gathering a quorum in order to execute an operation. A read operation needs to assemble a *read quorum*, an arbitrary collection of any r replicas or more. Similarly, to update an object, a *write quorum* of at least w replicas is required. Each replica holds a version number, a write operation reads the version numbers of w replicas, generates a number higher than any it has observed and stores the data at these replicas. The values of r and w are subject to two constraints: (1) $r + w > n$, and (2) $w > n/2$. The first constraint prevents reads and writes from conflicting and ensures that the most recent copy is read while the second constraint prevents two operations from occurring at the same time.

These techniques are robust and they remain consistent in the face of node and communication failures including network partitions. Different quorums for read and write operations can be defined and different weights, including zero, can be assigned to every copy. This form is called *weighted voting*.

In a mobile computing environment, the quorum consensus method will fail when most of the replicas are on mobile hosts; no quorum group can be formed since the availability of mobile hosts is not guaranteed. Also, the mobile hosts will not be able to perform transactions on the replicated data as to do so, they must form a quorum (if r is > 1 them even a read quorum cannot be formed).

2.4.1.4 ROWA

ROWA (*Read One Write All*) protocol [BG84] is a special case of quorum consensus method. For a *read* operation, the read quorum consists of only one node, but for a *write* operation, the write quorum consists of all copies.

ROWA reduces write availability by increasing the read availability. In a mobile computing environment, a mobile host can read replicated data (as opposed to a simple quorum consensus method since read quorum r is 1), but the problem of making a write quorum w is exacerbated. If more than one host becomes unavailable (mobile), the write quorum cannot be formed and no one will be able to perform any updates.

2.4.1.5 Available Copies

A variation of *ROWA*, known as *Available Copy Protocol*, is presented in [Goo83]. In this method a write operation updates only the copies that are available. The copies that are down are ignored. It increases write availability but in the face of network partitions an inconsistent state can be achieved when different partitions update different copies.

Since *Available Copy* method is a variation of *ROWA*, the read quorum can be formed and by making write quorum not dependent on those replicas that are maintained by nodes that are not accessible, the updates can also be performed. But since, the same is true for a mobile host, an inconsistent state will be achieved when different mobile hosts have updated different copies and there is no way to reconcile all of them.

2.4.1.6 Dynamic Voting

In *Dynamic Voting* [JM87] scheme each replica also maintains the update cardinality (apart from the version number). The update cardinality is the number of replicas updated by the last transaction. The *majority quorum* for a *read* or *write* is calculated with respect to the current update cardinality. The update is performed on available replicas. This update then in turn redefines the *majority quorum* to be more than half of

the *current replicas* rather than the whole system. The dynamic voting method can thus allow the system to adopt its quorum requirements according to state of the system.

Algorithms are presented so that a node can decide whether it is in the majority partition or not based on how many sites it can communicate with and the new quorums are computed accordingly. Operations are allowed to proceed only in the majority partition. This limit, in a mobile computing environment, puts the restriction on mobile hosts, not to update replicated data in disconnected mode.

2.4.1.7 Virtual Partitions

Virtual Partitions method was devised to increase the availability in network partitions arising from site/link failures [AT89]. The basic idea for each site to maintain a *view* consisting of the sites it believes it can communicate with. The operations can proceed only at nodes belonging to the *majority partition*, which is the partition that contains more than half of the nodes in the system. Views of sites change as site and communication failures occur. View management protocols are devised, based on two or three phase commit protocols, to ensure consistency of the views at all replicas. Each object is assigned read and write *quorums* which are redefined based on the current view to maximise availability, which ensures that quorums can be formed only in one of the partitions, namely, the majority partition.

Again, in a mobile environment, mobile hosts are not allowed to perform updates in a disconnected mode. Also when they connect, it is possible that they connect into a minority partition and thus are not allowed to perform updates.

2.4.2 Asynchronous Replication Protocols

The need to manage replicated directory systems give way for an asynchronous replication protocol. Many asynchronous protocols have been suggested. The asynchronous protocols are devised to overcome the overhead of computational and communication resource that is required by synchronous replication protocols. The asynchronous replication protocols achieve eventual consistency by propagating operations and resolving/reconciling conflicts. In this section, the well-known protocols are reviewed.

The asynchronous replication protocols show promise to support disconnected operation of mobile hosts. The asynchronous replication protocols are not dependent on the availability of hosts at the time of transactions, but at the time of propagation of updates.

2.4.2.1 Grapevine

In Grapevine [SBN84] an asynchronous replication protocol is presented to achieve eventual consistency. A node executes an update. A timestamp is associated with each updated item and then the node uses an unreliable multicast to propagate that update to all other nodes. During propagation the timestamps are used to determine the most recent version and only that version is retained.

Since an unreliable multicast is used to propagate updates, the reliability is compromised. To overcome this problem and ensure reliable delivery, copies of the databases are exchanged and merged periodically. Each comparison must span *all* copies in order to ensure reliability. However, the periodic updates impose a large load

on the network since each comparison involves sending a complete copy of the database to every other node.

In mobile computing environments, the mobile hosts communicate with the network for a limited amount of time, and the bandwidth is small. It is not possible for mobile hosts to send/receive large volumes of data to/from other nodes.

2.4.2.2 Global Name Service

Global Name Service (GNS) [Lam86], is a replicated naming database. In GNS, servers are organised in a logical ring. The updates are done at one copy and propagation is done through a *sweep* operation moving deterministically around servers. Periodically, the sweep operation visits every replica, collects a complete set of updates then writes this set back to every replica.

The sweep operation is expensive, since it has to collect all updates from all replicas, and cannot be executed frequently, hence, the speed of propagation is slow. Further, in case of failures, partitions or addition of a new replica, the ring has to be reformed from scratch. The reformation process must be controlled by an administrator, during which normal operation is suspended. That is, dynamic re-configuration is not supported.

The mobile computing environment is dynamic. Mobile hosts connect and disconnect frequently and this results in partitions. The system will spend most of the time reforming *sweep* ring, which is required whenever a network partition is created (resulting from connection/disconnection of mobile hosts) and normal operation will be suspended for a prolonged amount of time.

2.4.2.3 Epidemic Replication

In [Dem87], several randomised algorithms based on epidemiology theory are described for distributing updates asynchronously. They are intended to maintain a widely replicated directory or name-lookup database. Three epidemic communication methods are specified: *direct mail*, *rumour mongering* and *anti-entropy*. Direct mail propagates an update to other replicas using a single unreliable multicast datagram. A replica can use rumour mongering by selecting another replica randomly and sending it one or more *hot rumours*, again using unreliable datagrams. Hot rumours are recent update messages that the replica believes the other is unlikely to have observed. Several stopping rules are suggested to ensure that a message does not continue propagating forever, but none of the rules can ensure that a message has been propagated to every replica before stopping. Periodically, pairs of replicas perform a reliable exchange of database contents in an anti-entropy session, which ensures that updates will eventually propagate to all sites.

Conflict Resolution is done by associating each data item with a global unique timestamp (denoting the GMT) and the value with the latest timestamp is kept. The techniques can be combined, using direct mail or rumour mongering for fast, unreliable propagation, while anti-entropy provides a reliable backup if the other methods fail. However, anti-entropy is an expensive procedure since it involves comparing the contents of two copies of the database, one of them sent over the network, which could not be acceptable for large databases nor for large-scale systems. Further, combining several techniques means that a node can receive a message more than once, hence placing more load on the network.

In a mobile computing environment, *direct mail* message will not work most of the time as the mobile hosts connect only for a short duration of time and are usually not available. The *replica* originating *hot rumours* should not select a mobile host, as it will not be available. This will result in mobile hosts not receiving any *hot rumours*. Thus the only solution is periodical exchange of data contents, which is a heavy burden on the low bandwidth of a mobile host.

2.4.2.4 Wuu's Algorithm

In [WB84], a propagation scheme for maintaining a replicated dictionary using logs is presented. Each replica keeps a two-dimensional timetable (2DTT) recording what messages have been received by other replicas. The 2DTT allows the exchange of missing updates only rather than the whole copy of the database.

Periodically, each node i sends to every node j a message containing its 2DTT and all messages in its log that i believes j does not have. Node j receiving this message, extracts the new messages and updates its 2DTT by merging it with the incoming one. Although an attempt is made to send to a node message that this node has not received, redundant messages are still sent, as 2DTT is just an approximate view of what other nodes have. Another efficiency is that the 2DTT, of size n^2 , is sent as part of each message, which incurs high communication overhead especially for large scale systems. Further, the speed of propagation is very sensitive to the activation period.

In a mobile environment, the mobile host will not receive any of the messages. Since a mobile host connects for a short time, it might not be possible to receive any messages. It can certainly send its own updates (and adjust its view).

2.4.2.5 Time Stamped Anti Entropy

In [GL91], a weak consistency replication protocol is proposed, called the *Time Stamped Anti Entropy* (TSAE), which is based on the epidemic anti-entropy protocol [Dem87] and uses a set of data structures much like those in [WB84] to exchange updates. Each replica keeps a *summary vector* containing the latest message timestamps that it has received from other replicas. Periodically, a replica selects a partner at random and starts an anti-entropy session. First, they exchange their summary vectors then only the missing messages are exchanged. A best-effort multicast is combined with the anti-entropy scheme to speed propagation. The protocol ensures that the replicas eventually converge to a consistent state during normal operation and when recovering from failures. Reconciliation methods based on FIFO and total delivery orderings are provided.

Although TSAE attempts to reduce sending duplicate messages by exchanging states and then deciding which messages to send, still redundant messages are exchanged during normal operations. This is because a node can participate in several sessions concurrently and also because the status messages are out-of-date due to network delays. The amount of redundancy increases when TSAE is combined with the best-effort multicast. Also, the status messages exchanged place an extra load on the resources. Further, like other replication mechanisms, the scheme cannot scale to a large number of replicas since it is assumed that any node can communicate with any other node.

In mobile computing environment, these problems are exacerbated. The replica on a mobile host should not select a replica at random for an anti-entropy session as it might

result in requiring the replica on the mobile host to send more data than to receive. One solution is to select the same replica every time for the anti-entropy session (call it the base replica for that mobile host). This way the communication from the mobile replica to base replica will be minimal and mobile host will simply receive all the updates that were performed while it was disconnected.

2.4.2.6 Lazy Replication

Liskov and Ladin in [LL86] describe the *lazy replication* method, where updates take place at one replica, then replicas *lazily* exchanges new information through gossip messages. They rely on *multi-part timestamps*. Periodically, each replica sends a gossip message containing its timestamp and its message log to *all* other replicas and replicas are made consistent by merging the states and timestamps. Relying on a central node that generates the sequence order supports total causal orders.

However, *lazy replication* does not take full advantage of the information available in its timestamps and sends a complete copy of the message log in a gossip message which increases the communication and processing overhead. Also, it does not scale well, as every node has to communicate with *all* nodes. Further, it needs an extra log to prevent executing duplicate updates, hence, using excessive storage. Also, the scheme suffers from a slow speed of propagation due to the periodic updates.

In a mobile environment, a mobile host will have to wait for a reconnection before it can send a gossip message. The mobile host will miss-out all of the gossip messages that were generated while it was disconnected. The mobile host will heavily depend on the merging of states and timestamps. Due to the dependency on a central node for

generating sequence order, the mobile host will not be able to perform updates all the time (only when connected to central node).

2.4.2.7 OSCAR

A weak consistency replication mechanism is implemented by OSCAR (Open System for Consistency and Replication [DGP90]) using agents (a mixture of distributed and centralised elements). The architecture is based on two cooperating agents, called replicators and mediators.

Each node has a local replicator. For each update that is initiated at a node, the local replicator sends the message to all other nodes using an unreliable multicast.

There exists a master mediator, which periodically polls every replicator. A version vector for every database item from different replicas is received (polled). These version vectors are combined, and the result is multicast to every replicator.

The messages missed by a replicator are detected using the global version vector created by the master mediator. Each replicator asks other replicators for the missed messages. This global version vector is also used to purge a message from the log (no longer needed, no replicator will ask for it).

Two reconciliation methods are supported to resolve conflicts. These are based on FIFO (first in first out) order and overwrite order (latest timestamp wins).

A negative side of the scheme is that the mediator must contact *all* replicators, then, each replicator has to contact other replicators to extract the missing messages, which

causes large network traffic. Further, the mediator may become a bottleneck and prohibit scalability.

In terms of mobility, the scheme is not adaptable as presence of *all* replicators is necessary which is in direct conflict with the very nature of mobile computing environment.

2.4.2.8 Quasi-copies

In [ABG88], *Quasi-copies* method was presented to handle information retrieval systems. It is a weak consistency protocol. *Quasi-copies* are out of date replicas. These replicas are allowed to diverge in a controlled fashion by taking advantage of the application semantics.

The *quasi-copies* are managed by specifying predicates. These predicates can be based on time, version or arithmetic condition. Users define this management. For instance, a user can specify that the value of a copy should differ from the *real* value by no more than a constant, or that it should not be out-of-date by more than some period or number of versions.

The *quasi-copies* scheme relies on a central location, which is responsible for processing and propagating *all* updates. The central node will watch all updates and when a predicate is *about* to be violated, then propagation of updates is executed.

Although this approach allows for controlling the inconsistencies of the replicated data, it relies on centralised components, which cause bottlenecks in the system, limit the update availability and do not scale for large number of replicas. Also, the bound of

consistency given is only approximate since the possible occurrence of updates during the transmission delay from the central node to the quasi-copies is not taken into account.

In mobile computing environment, the predicates might be violated more often when a mobile host is not connected. Also, updates done on a mobile host are not available to the central node (and thus no checks can be performed to check the validity of predicates for controlling inconsistencies).

2.4.2.9 Epsilon Serialisability

A correctness criterion called *epsilon-serialisability* (ESR) is presented in [PL91b]. This criterion allows temporary and bounded inconsistency in replicas to be seen by queries. In [PL91a], an extension of *epsilon-serialisability* is used and an asynchronous replication approach is presented. The main emphasis is on transactions where several operations must be executed as a group.

The Operation semantics are used to increase the concurrency in the presented replica control methods. Four replica control methods are presented:

- Ordered updates method executes update transactions in the same order at every replica. It is assumed that a reliable and ordered message delivery mechanism is present.
- Commutative method is limited to commutative updates.
- Read-independent timestamp updates method supports operations that either produce immutable versions or overwrite older versions.

- Backward method is based on compensation, where operations are optimistically allowed to execute in any order, then rollbacks are used to undo the effects of transactions if inconsistencies are detected later.

Methods for bounding the amount of inconsistency seen by queries are presented and are based on ESR. The number of concurrent update-transactions with which the queries interleave measures the divergence.

For ordered updates, each query is given a global order number as if it is an update. Each time a query overlaps an update, an inconsistency counter is incremented. When the counter reaches a predefined limit, the queries are forced to run in total order. However, the details of the algorithm are left to the global ordering mechanism adopted.

For commutative updates, the transaction can acquire a certain number of read-write and write-write locks on objects – locks that are disallowed under strict serialisability. If a transaction attempts to acquire more conflicting locks than the limit, the transaction is blocked. However, locking is not a good technique for a large number of replicas [BK91].

Although updates occur at one replica and propagate asynchronously to other replicas, it is not specified how propagation occurs, nor does how eventual reliability is reached. However, the definitions of operation compatibility presented can be used to build conflict reduction mechanisms in weak consistency systems.

With mobility, the ordered updates method cannot be used, as mobile host is not available at all the time and a node is required to order all the update messages for that mobile host and send them to mobile host when it becomes available again. *Backward* method is more practical, where updates are allowed in any order and later on when mobile becomes available, the inconsistencies can be detected and the transactions can be compensated.

2.5 Related Research Issues

In the following sections, some mobile software systems are presented. These systems were designed to tackle software engineering or distributed database problems but were faced with the limitations of a mobile computing environment. Some relevant mechanisms are also being reviewed which attempted to solve the problems of mobility in a distributed computing environment.

2.5.1 Weak Operations and Weak Consistency

In [Pit96, PB95], a weak consistency model is presented for distributed databases. In this particular model, the transactions can originate both from mobile and stationary hosts. A database state is considered consistent if all of the integrity constraints are satisfied. In *weak consistency* model, instead of requiring maintenance of all integrity constraints, units of consistency, called clusters, are defined by partitioning the items of a database. This clustering is based on the location of data items, so that data in strongly connected sites belong to same cluster.

The consistency of a database state is defined in terms of consistency of each of its cluster states. A cluster state is consistent iff all intra-cluster integrity constraints hold.

A database is *d-consistent* iff all cluster states are consistent and all inter-cluster integrity constraints are *d-degree consistent*. The definition of *d-degree consistency* for an integrity constraint is dependent on its type. The cluster configuration is kept dynamic to take into account of dynamic nature of mobile computing environment, where clusters may be explicitly created or merged upon a disconnection or connection. Also a mobile host can move to different clusters due to its mobility.

The notions of *weak* and *strict* transactions are introduced. *Weak* transactions operate on *d-consistent* data items whereas standard read/write operations are strict. Weak transactions access data copies that belong to the same cluster. User transactions are decomposed into a number of weak and strict subtransaction units according to the degree of consistency required by the application. A *local-commit* (in associated cluster) and a *global-commit* (at reconciliation) are associated with each weak transaction.

The concept of *transaction proxies* is also introduced in [PB93]. These are used to take into account the vulnerability of the computation performed at mobile hosts. For each transaction executed at a mobile host, a dual transaction, called *proxy*, is defined. It will be executed on the support station of the mobile host. A proxy transaction may be considered as a subtransaction of the original transaction. Thus, any time a transaction is submitted to a mobile host its proxy transaction is submitted to its support station. Alternatively, proxy transactions may be executed off-line, when the network traffic or the load in the base station is low. In that sense, proxy transactions correspond to taking periodic backups of the computation, which is performed at a mobile host.

2.5.2 Currency Tokens

A mechanism for managing replicated file system in a mobile computing environment is introduced in [TD92]. An attempt is made to use asynchronous operations. The client determines the level of consistency required.

A primary-secondary server organisation is used. A client can write its update in its cache and continues execution. The primary node performs periodic pick-ups from the clients' cache and propagates the updates to the secondaries. A reliable communication mechanism is assumed to be available.

Two kinds of *reads* are provided by the mechanism. These are a *loose-read* and a *strict-read*. A *loose-read* reads from any copy without any guarantee concerning the value returned. A *strict-read* returns *the most recent value*. A *strict-read* requires synchronisation with at least majority of secondaries and with all caches (to satisfy *the most recent value* clause). Hence, it results in mediocre performance.

The updates (writes) are asynchronous, but to ensure consistency of all the read values, *every* write must be preceded by a *strict-read* to update a fresh copy of the file and to declare itself as a potential writer. This result in an asynchronous write preceded by synchronised access of the *strict-read*, which incurs large overhead and benefits of the asynchrony are lost.

To reduce the cost of *strict-read*, an optimisation is given by offering a *currency-token* to replicas that have just performed a *strict-read*. A client possessing a currency token can read consistent data from its local cache provided that there are no potential writers.

However, this optimisation is beneficial only if writes are rare since any write attempt will revoke the currency tokens and everybody is forced to *strict-read* again.

2.5.3 Coda

The Coda distributed file system [KS92] uses an optimistic replication protocol to manage distributed file services. It also provides a transparent file system interface to applications and supports disconnected operations without requiring any modifications to applications. This is achieved by allowing operations to occur asynchronously when network is partitioned. It uses the synchronous *Read One Write All* strategy during normal operation (no partitions).

Synchronisation among all sites is necessary to detect conflicts once the partition connects with rest of the network. It provides semantic-dependent rules for automatic recovery of specific applications (such as directories). The scheme aims at improving availability of the system in the face of partition. Optimistic replica control strategy is used to allow updates in any partition.

The architecture consists of a small number of servers on the fixed network and a number of clients. The replicas that are kept on servers are strictly coherent and are called first-class replicas. The replicas on clients are called second-class replicas.

Clients cache data on their local disks. This results in better performance and more importantly availability (while disconnected). A cache manager, *Venus*, is responsible for maintaining the cache and resolving any problems that occur due to disconnected operation (mobility). The cache manager can be in one of the following three states:

- **Hoarding.** The state of normal operation. The cache manager relies on the server and is in alert for possible disconnection. The client caches files from the servers and servers notify the client when the file is changed/updated by another client.
- **Emulating.** The disconnected operation is called the emulating state. If an object (file) is not in the cache, operations on that file appear as failures to application programs and users. An operation log is used to log updates while the client is disconnected.
- **Reintegration.** The cache of the mobile client needs to be resynchronised with the servers when the mobile client reconnects. The reintegration-state performs this vital task. Different strategies for handling file and directory updates are presented [KS93, KS95].

2.5.4 Isolation-Only Transactions

In Coda file system, the focus was on *write/write* conflicts. In [Lu96], *Isolation-Only Transactions* (IOT) is presented to address the problem of disconnected operation of mobile clients and taking into account of *read/write* conflicts.

The central idea of IOT model is imposing serialisability-based *isolation* requirements on partitioned transaction executions. Transactions executed on a disconnected client stay in a tentative state until the client regains connection to relevant servers. They are committed to the servers as soon as they pass consistency validation. Invalidated transactions are automatically or manually resolved to ensure global consistency. IOT is implemented under Coda file system.

2.5.5 Bayou

Bayou [Dem94, Ter94, Ter95, Edw97] is an infrastructure for supporting distributed and collaborative applications in which all user interactions involve reading and writing a shared, replicated database. This is done via Bayou Application programming interface (Bayou API) which supports two basic operations: *read* and *write*. It is a replicated, weakly consistent storage system. Bayou is designed to cope with varying network connectivity parameters. It works from high-bandwidth and constant connectivity to low-bandwidth and unreliable (may be occasional) connectivity. The later connectivity parameter fits with mobile computing environment.

In Bayou there is no single centralised location at which data is stored (all data is distributed). It relies only on pair-wise communications between computers, thus allowing the system to cope with arbitrary network connectivity. Bayou applications read and write to any available replica without explicit coordination with other replicas. A weak consistency among replicas is maintained. To handle the update conflicts that naturally arise in such a weakly consistent system, the applications are allowed to define their own semantic constraints. These semantic constraints are used to maintain the integrity of data and are also used to detect and resolve conflicts. The data update propagation is also governed by these semantic constraints.

Bayou servers manage replication. Each server holds a complete replica of the data. These replicas are weakly consistent, that is, at any one point in time different servers may have seen different sets of updates and therefore hold different data in their databases. Eventual consistency is reached by imposing a global order on write operations and by providing propagation guarantees. Each write operation carries enough

information so that a Bayou server can apply the writes in the correct order without coordinating with any other server.

Bayous' mechanisms for supporting application semantics fall into following categories:

- Application-defined conflict detection
- Application-defined conflict resolution
- Selection of session guarantees
- Selection of committed or tentative data
- Replica selection
- Selectable anti-entropy (data propagation) policies.

In Bayou a *write* consists of three components:

- Dependency Check
- Update Set
- Merge procedure

The *write* operation provides for the first two semantic categories. The dependency check is performed to insure that a write operation can proceed safely or not. If the dependency check succeeds, the update set is applied to the database (*write*). If the dependency check fails, the *merge procedure* is executed. It is intended to be an alternate update (*write*).

The *session guarantees* mechanism is used to establish a required level of consistency. An application can request for a certain level of consistency. This is to support different

applications that might have different requirements in terms of consistency. The following four session guarantees are supported by Bayou:

1. Read Your Writes ensures that the effects of any writes made within a session are visible to later reads within that session.
2. Monotonic Reads permits users to observe a database that stays up-to-date over time. It ensures that reads are only made to database replicas containing all writes whose effects were seen by previous reads within the session.
3. Write Follow Reads ensure that traditional read/write dependencies are preserved in the ordering of writes at all servers. That is, at every replica of the database, writes made during the session are ordered after writes whose effects were seen by previous reads in the session.
4. Monotonic Writes says that writes must follow previous writes within the session. In other words, a write is only incorporated into a replica's database copy if the copy includes all previous writes from that session, and the write is ordered after these previous writes.

If these guarantees cannot be met, then the calling application is informed.

Bayou also supports a mechanism that ensures a *stable write* on a server [Edw97]. A write becomes stable when the server has received and executed all writes that precede it. That is, when no new writes will ever be received by the server that will have to be ordered before that write. Any other write is called a *tentative write*. Tentative writes might need to be executed again if the server receives other writes, with earlier time-stamps. Since Bayou servers may receive writes in an order that is different from the

global order, servers may need to undo the effects of some previous tentative write and re-execute it. To increase the efficiency, it is desirable to stabilise updates quickly. To do this Bayou system designates a server as a primary. This primary server takes the responsibility of committing updates. Committed writes are ordered according to the times at which they commit at the primary and before any tentative writes. Each server maintains two views. In one view, only *stable writes* are reflected where as in the other view all (*stable* and *tentative*) writes are reflected.

Bayou also provides the facility to let application decide which replica will be used for its operations. This selection is made for the life span of an application. This can be used to optimise certain communication requirements. During an anti-entropy session servers of two replicas bring each other's databases up to date. Sets of known writes are exchanged. At the end the servers agree on the set of writes they have seen and on the order in which to perform them. Bayou is designed to support client-supplied anti-entropy policies.

Although the Bayou design does not include the notion of disconnected operation, the various degrees of connectivity and application defined semantic constraints can be used to adapt Bayou to support disconnected operation for a set of servers (network partitions).

2.5.6 Kangaroo Transactions

Kangaroo transactions [DHB97] are aimed at capturing the movement of a mobile host during the processing of a transaction. The transactions submitted by a mobile host are moved from one base station to another as the mobile host moves. In Kangaroo

transaction model, a mobile transaction *hop* from a stationary host to another, the state of the transaction and its progress is also moved.

Kangaroo transaction model uses the concepts of *Open Nested Transactions* [WS92] and *Split transactions* [PKH88]. The management of movement of a transaction submitted by a mobile host (called a mobile transaction) is the essence of kangaroo transaction model. Kangaroo transactions (KT) are discussed in context of multi-database systems in which each component DBMS is completely autonomous. Data in source systems (MDBMS) is accessed through a *Data Access Agent (DAA)*. DAA serves as a mobile transaction manager built on top of Global Database System (GDBS) and DBMS software residing on the fixed network Figure 2.6.

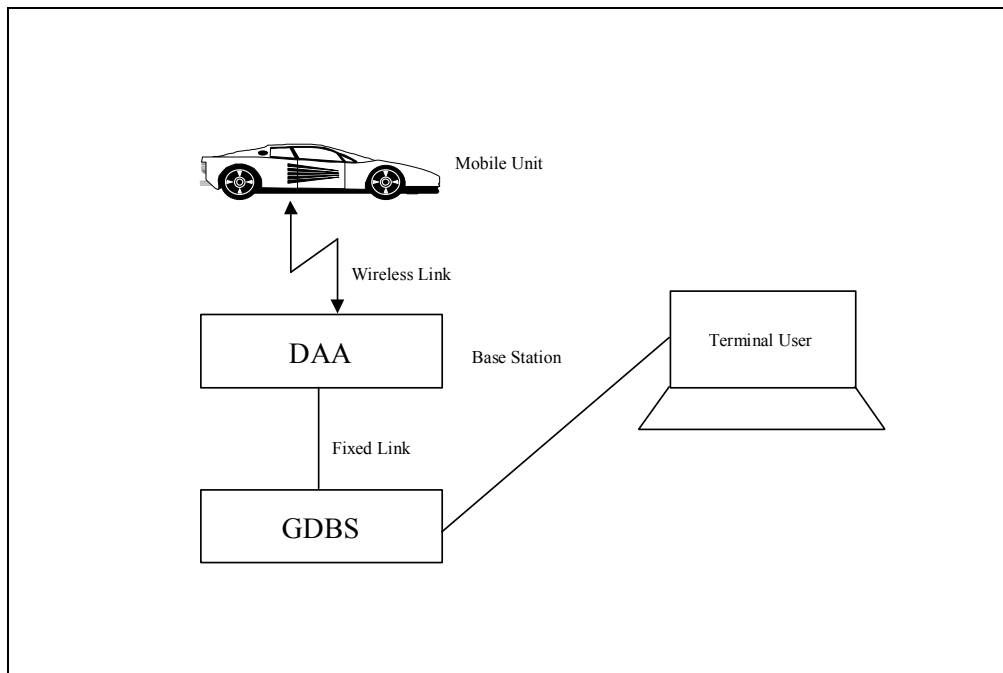


Figure 2.6 Relationship of Mobile Transaction Management to Multidatabase Systems [DHB97]

Two kinds of global transactions are considered:

1. Global transaction is composed of subtransactions, which can be viewed as local transactions to some existing DBMS.

2. Global transaction is composed of subtransactions, which can in turn be viewed as global transactions to other multidatabase systems. Thus giving a recursive definition to global transactions.

A mobile host starts a kangaroo transaction by making a transaction request to a DAA on a base station. The unit of execution on a base station is called a subtransaction named *Joey Transaction (JT)*. When control of a kangaroo transaction (due to the movement of mobile host) changes to a new DAA at another base station, the new DAA creates another JT. The old JT (on the previous DAA) is committed independently of the new JT. A linked list of all the Joey transactions of a Kangaroo transaction are kept to perform the undone operation due to some failure. This is accomplished by compensating transactions (Figure 2.7). Kangaroo transaction operates in two modes:

1. Compensating mode
2. Split Mode

These two modes are related to the two transaction models (Open Nested and Split respectively). In compensating mode, previously committed JTs will have to be compensated for. In Split mode (which is the default mode), previously committed JTs are not compensated for.

Kangaroo transaction model does not guarantee serialisability of kangaroo transactions. Locks are obtained and released at the local transaction level that in turn makes data visible to other transactions thus violating isolation. Atomicity is also not guaranteed in split mode as parts of a KT (JTs) are left committed on different previous base stations. The transaction processing support for disconnected mode is not discussed.

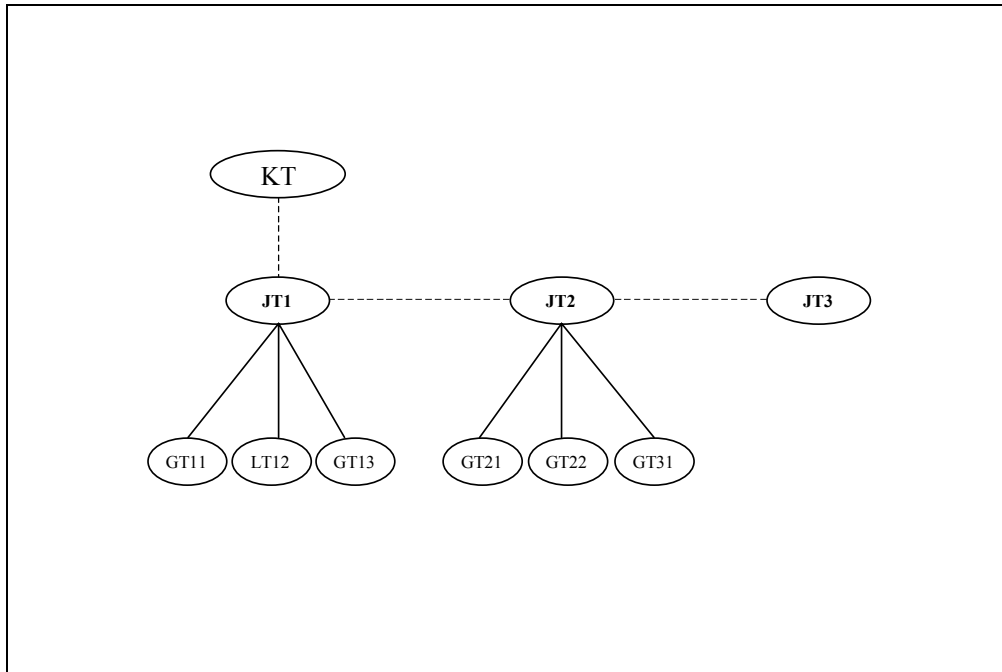


Figure 2.7 One Possible Kangaroo Transaction [DHB97]

2.6 Summary

Mobility exacerbates the tension between autonomy and interdependence that is characteristic of all distributed systems. The relative resource poverty of mobile elements as well as their lower trust and robustness argues for reliance on static servers. But the need to cope with unreliable and low-performance networks, as well as the need to be sensitive to power consumption argues for self-reliance.

Any viable approach to mobile computing must strike a balance between these competing concerns. This balance cannot be a static one; as the circumstances of a mobile client change, it must react and dynamically reassign the responsibilities of client and server. In other words, mobile clients must be adaptive.

To devise a transaction model/or a mechanism to handle disconnected operation of a mobile host, a number of transaction models and replication schemes are discussed in

the previous sections. A review of research being done to tackle this problem is also presented.

As discussed in previous sections, the transaction models cannot proceed when a mobile host is unavailable (that transforms a simple transaction into a long running transaction). Neither making a nested transaction, nor taking savepoints will help, because in either case the ACID properties are still maintained for the entire lifetime of a transaction. The transaction models that waive some of the ACID properties have no way of insuring data consistency in the face of network partitions (disconnected operation of mobile hosts, a controlled network partition). Additional structuring of the transactions does help to organise normal processing, but it does not contain the necessary semantics to cater for long running transactions, which result from non-availability of mobile hosts.

A mobile host will typically operate in three modes (states):

1. **Normal.** Connected to network and participating in transactions.
2. **Disconnected.** Not connected to network and running transactions on its local copies.
3. **Reconciling.** In this mode, the mobile host connects to the network and tries to resynchronise itself with the rest of the network.

These three modes are recognised by [KS92, section 2.5.3] and are named as data hoarding, disconnected operation and reintegration respectively.

Replication of data is considered to be a viable mechanism to support disconnected operation of mobile hosts but these protocols are handicapped by the fact that they were

designed to operate in an environment where network partitions are not so often (as opposed to a mobile computing environment). The synchronous protocols require a certain replica (or certain number of replicas) to be available all the time. The shortcomings of different asynchronous replication protocols are also presented.

The next three chapters present the twin-transaction model that provides the mechanisms to support connected/disconnected modes of operations of a mobile host. Chapter 3 discusses various components of the twin-transaction model and explains the consistency model. Chapter 4 discusses the transaction execution mechanism and the transaction history log maintenance. The determination of probability of success/failure is also discussed in in this chapter. The twin-transaction model relies on transaction history logs to maintain consistency among different hosts. A resynchronisation process is executed each time a mobile host connects and it maintains the global consistency of the system. The synchronisation process is discussed in detail in chapter 5. The twin-transaction model also computes probabilities of success/failure of transactions. These probabilities are used to optimise the synchronisation process by heuristically reducing the number of transactions that might be in conflict as discussed in chapter 4.

Chapter 3

Twin-Transaction Model

The Twin-Transaction model is presented in this chapter as a model to support mobile data access. This chapter is derived from [RZ96a, RZ96b, RZ97, RZ98 and ZR96]. The Twin-Transaction model is designed to fulfil the requirements of a mobile computing environment. The support for transaction execution in disconnected mode of operation of a mobile host was the driving force in some major design decisions. These design decisions are discussed first before presenting the model itself. The Twin Transaction model relies on replication of data and intelligent management of transaction execution, logging and resynchronisation to support connected and disconnected modes of operation of a mobile host. The support for these mechanisms is necessary to maintain consistency of data. The transaction execution and resynchronisation are mentioned in this chapter and are discussed in detail in chapters 4 and 5.

3.1 Design Decisions

A mobile computing environment is characterised by the fact that some hosts are mobile and some are fixed. The disconnected mode of operation of a mobile host is a special case of network partitioning (mobile host in one partition and rest in the other). The sharing of data in such an environment is a challenging problem. To develop a new transaction model/mechanism to support such a dynamic environment, lets first present the design objectives and decisions that were taken during the research to meet those

objectives. Then the Twin-Transaction Model (TTM) is presented and is shown that its design objectives are met.

A mobile host is either connected or disconnected. The transaction model must be able to support both modes of operation. Also the transition from disconnected mode to connected mode must also be handled. This transition is called *Resynchronisation*. Thus a mobile host works in three different states as shown in Figure 3.1:

- Connected
- Disconnected
- Resynchronisation

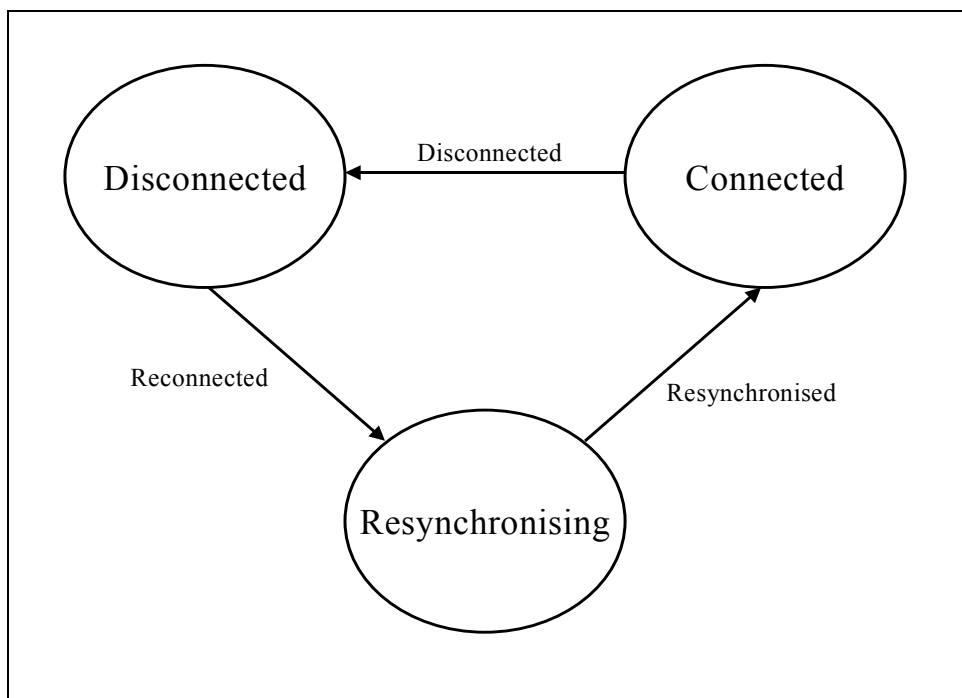


Figure 3.1 Mobile Host Execution States

The support for disconnected mode is necessary. This will provide a mobile with continued data access and continued transaction execution in the presence of network failures (disconnection). The mobile host, in disconnected state, will become a self-reliant server and all of the data requests will be furnished by using the replicated data

locally. The work done is synchronised with the rest of the network on reconnection during the resynchronisation phase.

Any model for disconnected operation of mobile hosts must provide mechanisms to support the following:

- Disconnected execution of transactions.
- Detection and resolution of conflicts that arise due to the disconnected execution of transactions.
- Reduce the number of conflicts arising due to the disconnected execution of transactions.

The detection and resolution of conflicts is necessary to maintain the consistency of the data. The data becomes inconsistent due to partitioned sharing. That is, execution of transactions on both mobile host and fixed network (in disconnected state). Transaction model should detect the following type of conflicts:

- write/write, and
- read/write

In *write/write* conflict, the data (object) is updated on both a disconnected client and on the corresponding servers.

In *read/write* conflicts, the data (object) is updated in one partition (mobile host or corresponding server) and is read in another.

In current state of practice, replicated systems only detect *write/write* conflicts. Partitioned read/write sharing is an important issue in a mobile computing environment. Undetected read/write conflicts pose a serious threat to data integrity (as mentioned in section 1.2.1) and may impede the useability of disconnected operation.

3.1.1 Disconnected Operation

Transaction model should be aimed to support disconnected operation of mobile hosts and provide mechanisms to maintain data in a consistent state while allowing *read/write* and *write/write* partitioned sharing.

The disconnected operation is inherently requiring optimistic design approach to data sharing. The pessimistic approach will evidently restrict all access to data in disconnected operation. Thus the transaction model must employ optimistic approach to data sharing.

The optimistic approach of transactions model should assume that all transactions are consistent in disconnected state. These transactions should be validated against the transactions executed on the server and invalidated transactions should be resolved which can result in re-execution, abortion or compensation of transactions. The results, produced by transactions in disconnected mode, should be made available to all other transactions that will execute afterwards. The inconsistent transactions (that produced an inconsistent result or operated on an inconsistent result) executed in disconnected mode must be detected.

3.1.2 Inconsistency Detection and Correctness Criteria

Mobile hosts need to operate in disconnected mode for various reasons, but without a proper inconsistency detection mechanism the consistency of the system will be compromised. In traditional transaction management systems, a concurrency control mechanism is used to avoid the inconsistencies by detecting sharing conflicts. The typical correctness criterion used in these concurrency control algorithms is serialisability. In advanced transaction models, on top of concurrency control mechanism, semantic information is used to allow certain sharing conflicts.

The disconnected mode of operation of a mobile host makes the concurrency control mechanism complex. The sharing conflicts cannot be detected between partitions of a network. This will, in return, make the data inconsistent unless some measures are taken to detect these data inconsistencies and reconcile the conflicts (possibly by re-execution, compensation, or abortion of conflicting transactions).

Application-specific definition of consistency can be used to model the consistency criterion of disconnected operation. This has been used in Bayou system [Dem94]. The advantages of such a scheme are that consistency requirements for different applications can be different and the accuracy of detection of inconsistencies will be perfect. The inconsistent access to data is detected and inconsistency is resolved. The main disadvantage is that semantic knowledge for all disconnected operations is required.

Another scheme would be that the transaction manager should maintain ordering information among all data access operations. This way during reconnection, all the conflicts can be detected and inconsistent transactions can be resolved.

The twin-transaction model employs a mixed approach. The consistency validation is done using the transaction ordering. The twin-transaction model also employs some heuristic to detect a conflict that will cause inconsistency upon reconnection. Application-specific conflict detection can also be used to detect possible conflicts in disconnected mode. The application is given the control to decide whether to *twin-commit* a transaction which might be in-conflict according to some heuristic.

This way the twin-transaction model aims to:

- Maximum number of transaction to proceed, and
- Heuristically detects inconsistent transactions, thus minimising the amount of work done during reconnection.
- Provide status information to applications issuing a transaction. This status information can be used as a feedback to a transaction.

3.2 The Model

Twin-Transaction Model consists of a number of components. These components work together to meet the goals defined in section 3.1. The model defines transaction execution mechanisms to cater for connected and disconnected modes. The defined resynchronisation mechanism achieves a consistent state on reconnection of the mobile host.

The twin-transaction model utilises heuristics to determine the conflict probability of a transaction that is being executed. This heuristic information is also made available to applications.

Figure 3.2 shows the skeleton of the twin-transaction model. It consists of following major components.

- Twin-Transaction Manager (TTM) that handles all the transactions.
- A number of agents/modules to perform necessary tasks to keep the system consistent.

Both server and client components of the system use the same architecture. The difference is in the roles that they perform and in the definition of a data item that is contained in the replicated database (section 3.2.2).

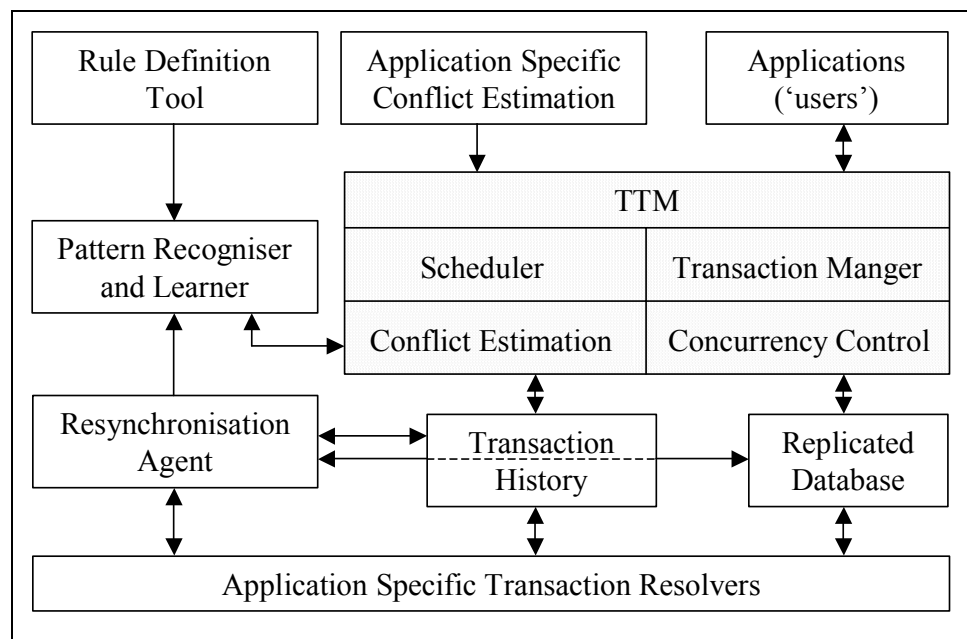


Figure 3.2 Twin-Transaction Model

The conceptual blocks of *twin-transaction model* are shown in Figure 3.2 The TTM (twin-transaction manager) handles concurrency and execution of a transaction. It also estimates the conflicts that are possible. The conflict estimation is performed using application specific conflict estimation and heuristics employed by pattern recogniser and learner. The executed transaction is kept in a transaction history database until the point when resynchronisation process has completed and verified the transaction.

Application specific resolvers are executed during the resynchronisation process to maintain data/transaction consistency.

This section discusses the building blocks of Twin-Transaction Model.

3.2.1 Transaction Management

A transaction manager has two jobs - to *mediate* the execution of individual requests, and to support specific transaction management functions, such as commit or abort. *Mediation* means determining which request will be allowed to execute, and triggering actions that make the execution safer.

The Twin-Transaction model implements transaction management using a set of *mobile transaction managers (MTM)* plus a *fixed transaction manager (FTM)*. The MTMs are responsible for handling transaction requests on each of the mobile hosts, while the FTM handles requests from hosts on the fixed network and from the MTMs. A set of *global reconciliation algorithms* is used between MTMs and FTM to detect any conflicts/inconsistencies that may arise and resolve them. The transaction management structure is depicted in Figure 3.3.

The users of twin-transaction managers (FTM and MTMs) can be end-users, programs, etc. Diversity is accommodated by diverse algorithms/heuristics to handle transactions at each MTM; the algorithms/heuristics at FTM are fixed for all the fixed hosts, while each of the MTM defines its own method to reconcile with the FTM. The FTM does not make any other distinction between fixed and mobile hosts.

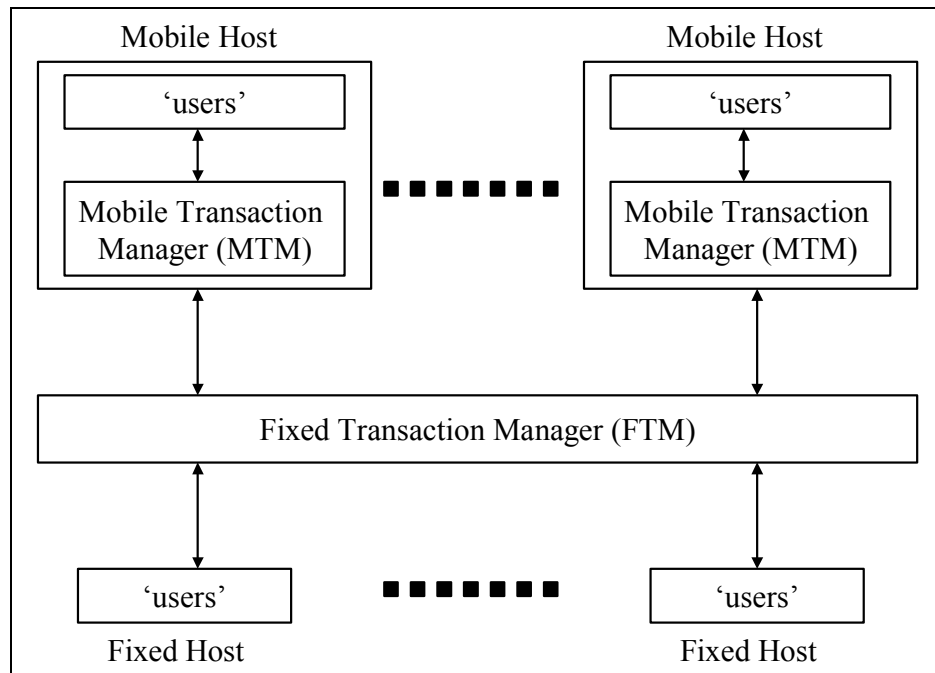


Figure 3.3 Twin-Transaction Management

Unlike classical transactions, the twin-transaction model has no simple correctness criterion. Each MTM decides what is a correct behaviour. The *correctness* of execution of transactions is relative to these individual behaviours. The consistency of FTM (and MTMs) is *tentative* most of the time (unless all the MTMs are connected and have gone through the reconciliation process).

3.2.2 Data Item Structure

The twin-transaction model relies on replication of data items to support disconnected operation. A replicated data item X is encapsulated into a set, called X_{set} , that defines the number of replicas of X and the sites where each replica is maintained. Even when a data item is not replicated, it is still encapsulated into X_{set} . Each data item on the two types of twin-transaction managers (FTM and MTM) is defined differently.

DEFINITION:

On FTM, a data item is represented by a X_{set} , which is defined as:

$\{ X \}$, maintained by an FTM, or

$\{ X_1, X_2, \dots X_n \}$, maintained by an FTM and $n-1$ MTMs.

DEFINITION:

On each MTM, a data item is represented by a X_{set} , which is defined as:

$\{ X \}$, maintained by MTM, or

$\{ X_{\text{MTM}}, X_{\text{FTM}} \}$, maintained by an FTM and that MTM.

The granularity of data item X is left unspecified.

The MTM is only aware of replicas on FTM. Only FTM is fully aware of all the replicas of a data item. Thus FTM has the responsibility to maintain consistency and incorporate correctness criteria for the distributed mobile system. Each MTM is only responsible for maintaining their local consistency and correctness (autonomy) while operating in disconnected mode.

3.2.3 Transaction Structure

The twin-transaction model relies on the resynchronisation process and on the transaction conflict resolvers to keep data consistent. To incorporate the resynchronisation process, a transaction in the twin-transaction model is defined using a *transaction script*. A *transaction script* is a twin-transaction that performs required operations and optionally caters for situations that might arise during the resynchronisation process (eg. transaction is re-executed).

A *transaction script* describes control flow and other execution strategies of a twin-transaction. The transaction script consists of *steps* that define the basic units of work in a transaction. Control flow between steps can be modelled by the usual elements of

- Sequence
- Branch
- Loop

The twin-transaction model defines a *twinning* process, which is applied to each transaction. The application of *twinning* process to a transaction creates two transactions, called *twin-transactions*. Thus, for a transaction T , two twin-transactions T_α and T_β are created (thus the name Twin-Transaction Model). The twinning process is applied in both *connected* and *disconnected* modes of operation. The α -*twin* of the transaction is executed locally and β -*twin* is executed on FTM. Detection and resolution of conflicts is performed using both *twins* of a transaction.

The *twinning* process transforms a transaction into its α and β twin-transactions. The twinning of a transaction is implicit and is done for each transaction. Explicit *twinning* of a transaction is also allowed, where the application submitting a transaction can submit the two twin-transactions (α and β).

The *transaction script* implemented by a particular implementation of the twin-transaction model must include ways to define α and β twins of the transaction.

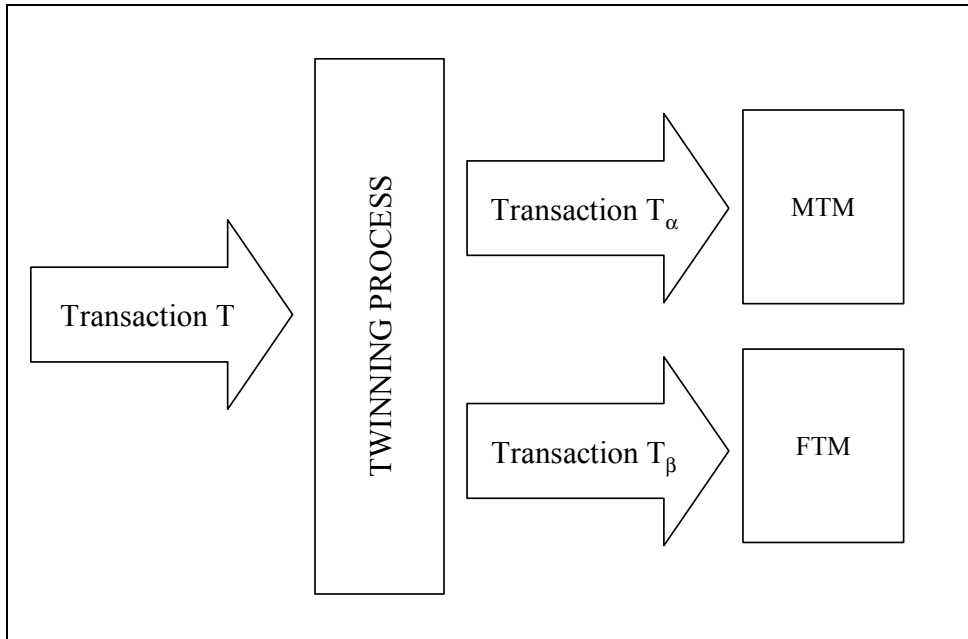


Figure 3.4 The Twinning Process

The two twins of a transaction go through a number of states before reaching their final state. The transition of states is dependent on connected or disconnected mode of operation of the TTM (Twin-Transaction Manager, FTM and MTM). These states and their valid transitions in connected and disconnected modes are shown in section 4.1.

3.3 Resynchronisation

The resynchronisation process is necessary to maintain the consistency of data. The local state of MTM and local state of FTM evolves along their own courses (different transactions take them to different states from an initial consistent state). The resynchronisation process is responsible to combine the two local states and make data consistent. To achieve this goal, the resynchronisation process needs a record of transactions that are executed on both MTM and FTM. This record is commonly known as the *Transaction History* or *Transaction Log*. Apart from transaction history log, the transactions in *pending* state on MTM and FTM are also needed to ensure that no inconsistent data access was allowed.

The resynchronisation process is executed whenever an MTM connects with the FTM and information exchange takes place (*pending* transactions and their transaction execution history). On re-connection, the resynchronisation process will make sure that replicated data items on MTM reflect the updates done on FTM.

The resynchronisation process can be divided into following main tasks and are discussed in section 5.1.

- Propagation of *resolved* transactions from FTM to MTM to transit transactions from *tentative-commit* to *commit* state.
- Propagation of *pending* transactions from MTM to FTM for resolution.

The transaction history management is discussed in detail in section 4.2 and the resynchronisation process is explained in chapter Chapter 5.

3.4 Consistency Model

The ACID properties of a transaction (mainly durability) are related in *twin-transaction model*. The consistency of database is guaranteed if a transaction transforms a database from one consistent state to another consistent state. In the twin-transaction model, transactions go through different states before reaching their final *resolved* state. Among these states, the states *pending* and *resolved* are important to consistency management. Since after *pending* state the results of the α -*twin* are made visible locally and after *resolved* state the results are made visible globally. Thus consistency of database must be maintained locally and globally. That is, the transactions executed on an MTM must be consistent before they are checked against global consistency (if they are not

consistent locally, they will not be consistent globally). Thus to achieve consistency, the twin-transaction model employs two layers of consistency model, *local* and *global*.

3.4.1 Local Consistency

The *local* consistency level requires that α -*twin* of a twin-transaction on a TTM must be locally serialisable (LS) with other pending or ongoing transactions executed on the same TTM. LS ensures that local interleaved transaction executions are always equivalent to a serial execution, which is necessary for the pending transactions to present a locally consistent view about their results.

3.4.2 Global Consistency

Since each TTM executes transactions locally and these transactions are resolved at a later stage, an optimistic concurrency control mechanism to maintain consistency is the natural choice. The global consistency model is defined by the resynchronisation process, which defines how transaction histories are to be merged.

The twin-transaction model defines two levels of global consistency models, ie, two types of resynchronisation mechanisms (two ways to detect conflicts during resynchronisation of histories). Both the models are serialisability-based to define admissible disconnected operations. These two models are also used in IOT model (Isolation Only Transaction model) [Lu96].

In the *twin-transaction model*, it is left to the application to decide which level of consistency model is required. The two levels are:

- Global One-Copy Serialisability (G1SR)

- Global Certification

These two levels of consistency model are described below.

Global One-Copy Serialisability (G1SR)

If a disconnected transaction T 's result is copied to the server as is, T must be 1SR (One-Copy Serialisable) with all previously committed transactions [Dav82, Dav84a].

G1SR consistency criterion ensures that results of disconnected transactions can be simply propagated to servers as if the transactions are ran in a non-replicated environment. It ensures that the resultant transaction history is equivalent to some serial execution of all involved transactions.

G1SR is a natural adaptation of the traditional 1SR model to an optimistic replication environment. It is suitable under the following conditions:

- Duration of disconnection is short.
- Distributed computations (most of them) are concurrent.

Example: Suppose a transaction T_1 is executed on an MTM, which reads data item X and then updates data item Y at time t_1 . Also suppose that another transaction T_2 is executed, that updates data item X at time t_2 .

Under G1SR, the transaction T_1 will be accepted as it can be serialised before transaction T_2 . But suppose that transaction T_1 was supposed to be operating at the most

recent value of data item X and $t_1 > t_2$, then the result produced should not be accepted, or at least the MTM that executed T_1 should be informed of the state of acceptance.

To overcome this limitation of G1SR, a stronger consistency criterion can be adopted and is defined below.

Global Certificates (GC)

Global certificates require that if a disconnected transaction T 's result is copied to the server as is, T must be not only *serialisable with* but also *serialisable after* all the previously committed transactions [Lu96]. GC can address the problem discussed earlier by invalidating transaction T_1 because it cannot be serialised after T_2 .

Essentially, Global Certificates criterion assures that the data accessed by a disconnected transaction is unchanged on the servers during the disconnection, thus it represents a disconnected computation that is the same as if the client had not disconnected. It is a very strong consistency requirement.

3.5 Conflict Resolution

The conflicts among *local* transactions (α -twins of transactions) are detected and resolved during transaction execution (classical transaction model). The conflict detection and resolution in a global consistency mode requires a completely different approach. This is due to the fact that transactions are executing in disconnected mode and conflicts among transactions running at two (or more) distinct TTMs are not known until they connect and exchange information (resynchronisation process). The *twin-*

transaction model keeps history logs to detect conflicts and resolve those conflicts during the resynchronisation process.

The resynchronisation process is dependent on the resynchronisation relation \Leftrightarrow defined in section 4.2. This resynchronisation relation defines how transaction histories are to be merged together. This resynchronisation can be application specific, ie, application can define its own resynchronisation process which can possibly use semantic knowledge about the application to accept a wider range of transactions and number of conflict can be minimised.

When a conflict is detected, the transaction manager has to take steps to resolve the conflict. A conflict can be resolved using one of the following methods:

- Re-execution
- Abort
- Application Specific
- Human Intervention
- Ignore (No-Conflict)

These methods and their applicability to the twin-transaction model are discussed in the following sections.

3.5.1 Re-execution

When a conflict is detected that is caused by a transaction T , it can be resolved by re-executing transaction on up-to-date data if the transaction belong to the *re-execution* class of transactions. This conflict resolution method is used in situations where input

from users/applications is not required and the transaction in itself makes all the necessary decisions to perform operations on the up-to-date data. For example in a banking environment a deposit of \$100 to an account can be split into three operations:

- Read balance of Account.
- Increase balance by 100.
- Write new balance of Account.

If the transaction contains all three operations it can be re-executed. On the other hand if operation 2 is not part of transaction (its not a READ or WRITE operation) the transaction cannot be re-executed, as new balance cannot be calculated.

To increase the number of transactions that can belong to this *conflict-resolution-class* (CRC, section 3.5), a script-based transaction manager is required. A transaction submitted as a script, along with all the data access operations can be re-executed, since it only involves re-execution of the transaction script (application control is built into transaction script).

The main advantage of this option is that it can be performed automatically without putting additional demand on the transaction programmers (accept that they must define transaction in the scripting language of the transaction manager). This ability to restore consistency automatically during the resynchronisation process by re-executing transaction independent of application provides much desired transparency. The disadvantage is that a considerable amount of work will be done/re-done by the resynchronisation process and the amount of data transferred will also increase (transaction becomes bigger as it now contains the transaction logic too).

In the twin-transaction model, re-execution essentially translates into executing β -twin of a transaction during the resynchronisation process. The twin-transaction model provides means to define α and β twins of a transaction to be different (explicit twinning). Since α -twin is not re-executed, it need not contain any transaction script, only β -twin should contain a transaction script to help synchronise the transaction. This way application specific semantic knowledge of a transaction can be applied to resolve a conflict by simply executing β -twin. Explicit use of application specific knowledge to resolve conflicts is also an option and is discussed in section 3.5.3.

3.5.2 Abort

It is the worst conflict resolution option and it dictates to automatically *abort* a conflicting transaction by throwing away the results produced by it. It is similar to the *rollback* mechanism of classical transaction recovery.

A transaction that cannot be resolved by any of the defined conflict resolution methods is aborted. It is independent of application. The originating TTM can (if indicated by the transaction) inform the transaction originator about the result, which in turn can take appropriate actions.

The main drawback of this approach is that transactions that are dependent on the results produced by this transaction (that is aborted) must be resolved (possibly *re-executed*) again.

For example, a manager and his secretary can make appointments in a diary (database). If manager is making appointments while being disconnected and secretary is also making appointments, then it is possible that two appointments can be made for the same time slot. During resynchronisation, this conflict will be detected and one of the appointments must be aborted. In this particular case, the originating application (secretary diary manager, or manager diary manager) must be informed of the abort decision.

3.5.3 Application Specific

The third and most flexible option is to invoke a user-supplied *application-specific* program whenever a conflict is detected. A transaction can specify a program that should be executed for conflict resolution if that transaction is later found to be in conflict during resynchronisation process. The *application-specific* conflict resolution has a number of advantages over the two previously discussed methods (ie Re-execution and Abort). These advantages are:

- Semantic knowledge of application is integrated into transaction management.
- Automatic compensation of disconnected transactions.
- Semantic knowledge utilised to resolve conflicts.
- Application specific consistency validation.

The application specific conflict resolution (ASCR) makes use of semantic knowledge of application to minimise conflicts and maximise number of transactions that are accepted by the transaction manager. By default, consistency validation is performed solely on the basis of syntactic information of data access operations. But this is a

conservative approach. An invalidated transaction does not necessarily mean that the data involved are actually inconsistent (it simply means that it can be inconsistent). The ASCR mechanism provides a mean to fine grain the check of data inconsistency using application semantics.

Consider the diary application discussed in the Section 3.5.2. In place of simply aborting one of the appointments, an application specific resolver can be implemented which can test whether the other appointment is valid or not. There are a number of cases that can be considered by the ASCR to see if the transactions are actually in conflict. These can be:

- Check if the appointment is with the same person?
- Check if the appointment is with someone from the same company?

Possibly a conference.

Once decided that the transactions are in conflict, the ASCR can execute some compensatory actions to undo the effect of the aborted transaction. Then ASCR can make another appointment for the person whose appointment has been cancelled and send a fax to that person about the change of appointment and ask for a confirmation. In a nutshell, rather than aborting/re-executing transaction syntactically, the semantics of the transaction are checked and if in-fact it is in conflict, then the effects of transaction are removed and a contingency transaction (to make another appointment) is executed.

In an environment where disconnected operations/transactions are executed more often (mobile computing environment), application specific conflict resolution is more valuable as compared to traditional optimistic computation models.

3.5.4 Human Intervention

There might be certain class of transactions for which none of the above mentioned methods can be incorporated. These transactions can choose to be resolved by human intervention manually. The human intervention is also required in cases where an application resolver fails to automatically resolve a transaction. The users will be requested to manually repair the transaction.

3.5.5 No-Conflict

To cater for those transactions that do not have any impact on the global state, a transaction can belong to No-Conflict class. Transactions belonging to this class are of two types:

- Transactions whose results are not propagated to FTM.
- Transactions whose results are always propagated to FTM.

3.6 ACID Properties

The ACID properties (*Atomicity*, *Consistency*, *Isolation* and *Durability*) are traditionally satisfied by the transactions. Concurrent transactions are executed in some serial order and effects of successful transactions are made permanent while effects of other transactions are removed.

In the twin-transaction model, a transaction goes through a number of states before reaching its final state. The twin-transaction model is designed to handle disconnected operations and make data available in the face of network partitions (disconnected mode). The *twin-transaction model* does not maintain ACID properties of transactions due to the following reasons:

- Transactions are kept in *pending* state.
- The results produced by a transaction in *pending* state are made visible to other transactions.
- The transactions produce results that are *tentative* until the transactions on which these results are dependent are reconciled/synchronised.
- A transaction in *pending* state is resynchronised and reaches its resolved state during resynchronisation process. The transaction might be aborted/compensated during this process.

Atomicity

Atomicity is guaranteed at each TTM. Either all or none of twins of a *twin-transaction* are executed during system crashes. It is also maintained to the extent that all twins of a *twin-transaction* will eventually reach their resolved state. The two twins are executed separately and the results are synchronised at a later stage.

Consistency

See section 3.4.

Isolation

The *Twin-transaction model* does not make intermediate results of transactions visible and guarantees the *Isolation* property for transactions.

Durability

The classical durability property requires that once a transaction completes successfully, its results must be able to survive. This also implies that once the results of a transaction are made available (to other transactions) it must remain a permanent part of the system-state until modified by later transactions. In the *twin-transaction model* durability of the transactions is only guaranteed when they have been *resolved* (classified as successful completion of a transaction). This is due to the fact that a committed transaction (in

pending state) on a TTM might be aborted/compensated during the resynchronisation process.

3.7 Twin Transaction Specification

In *twin-transaction model* each transaction goes through a twinning process and generates two twins (α and β). These two twins are executed in two different environments (Local and Global), thus requiring two consistency models (Local and Global, section 3.4). The two twins (α and β) need not be identical, especially when explicit definition of both twins is provided by the application submitting a transaction.

In the twin-transaction model each transaction belongs to a certain *conflict-resolution-class (CRC)*. These classes (CRCs) are defined based on the conflict resolution method. When a conflict is detected, the class of the transactions in conflict determine how the conflict should be resolved, ie, which conflict resolution method should be used.

The α -*Twin* of a twin-transaction is executed in a local environment and its schedule is guaranteed to be locally serialisable. For the β -*twin*, before execution of a transaction starts, the transaction must specify its *conflict-resolution-class (CRC)*, section 3.5) that will be used to resolve any conflicts during the resynchronisation phase. The CRC of each transaction can optionally include an application specific conflict resolver (section 3.5.3). Apart from CRC, the transaction must also specify the global consistency model to be used (section 3.4.2).

A transaction can belong to more than one CRC. In such a case a default sequence of conflict resolution methods is adopted:

1. Application specific conflict resolver
2. Re-execution
3. Abort

This default sequence can be altered by the transaction specification and can include *Human Intervention*.

Thus the complete specification of a twin-transaction include the following:

- The transaction
- Conflict resolution class
- Global consistency model, and optionally
- Application specific conflict resolver
- Conflict resolution method sequence

3.8 Summary

The chapter explained the objectives of a transaction model for mobile computing environment and presented the Twin-Transaction model. The *twinning* process is introduced that generates two twins of a transaction (α and β). The state transitions of a transaction (the two twins) are explained under the connected and disconnected modes of operation. The two consistency models supported by twin-transaction model are explained.

The next chapter discusses transaction execution in connected and disconnected modes. The concurrency control mechanism adopted and the determination of conflict probability are also described.

Chapter 4

Transaction Execution and Management

This chapter explains the transaction execution process in connected and disconnected modes of operation. The mechanisms to maintain and manipulate transaction history are discussed. The concurrency control alternatives are discussed in light of mobile computing environment. The probabilistic calculation of success/failure of a transaction is discussed in detail.

4.1 Transaction Execution

The structure of a transaction in the twin-transaction model is explained in section 3.2.3. A transaction in the twin-transaction model goes through the *twinning* process and generates two twin transactions (α and β transactions). The two twins of a transaction go through a number of states before reaching their final (commit/resolved) state. The transition of states is dependent on connected or disconnected mode of operation of TTM (Twin-Transaction Manager, FTM and MTM). These states and their valid transitions in connected and disconnected modes are depicted in Figure 4.1 and Figure 4.2 respectively.

The twin-transaction execution is different in connected and disconnected mode. In the disconnected mode:

- The *twinning* process generates two transactions T_α and T_β .

- Transaction T_α (α -twin of transaction T) is executed and results are committed (locally). Transaction T_α is placed in a *tentative-commit* state.
- Transaction T_β (β -twin of transaction T) is placed in *pending* state.
- On reconnection transaction T_β is reconciled with FTM. It is placed in *resolving* state.
- On FTM transaction T_β reaches *resolved* state when all MTMs have resynchronised or time-range Δt (section 4.2) have elapsed.
- Transaction T_α reaches *committed* state when transaction T_β reaches *resolved* state.

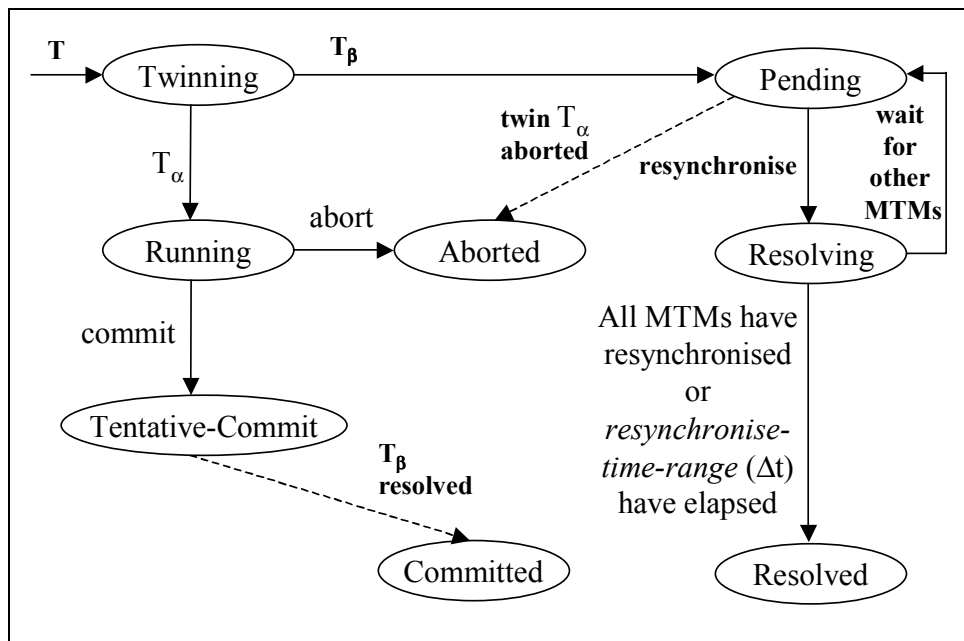


Figure 4.1 Twin-Transaction states in disconnected mode

In the connected mode:

- The *twining* process generates two transactions T_α and T_β .
- Transaction T_β is submitted to FTM for execution. On FTM it goes through the following process:

- The *twinning* process generates α and β twins of transaction T_β . The MTM that submits transaction T_β explicitly defines α and β twins to be semantically equivalent to α and β twins of original transaction.
- The α -twin of transaction T_β is executed. It is placed in a *tentative-commit* state and results are committed on FTM.
- The β -twin of transaction T_β is placed in *pending* state.
- Transaction T_β reflects the state of its β -twin.
- On FTM β -twin of transaction T_β reaches *resolved* state when all MTMs have resynchronised or time-range Δt (section 4.2) have elapsed
 - Transaction T_α (α -twin of transaction T) is executed and results are committed (locally). Transaction T_α is placed in a *tentative-commit* state.
 - Transaction T_α reaches *committed* state when transaction T_β reaches *resolved* or *committed* state.

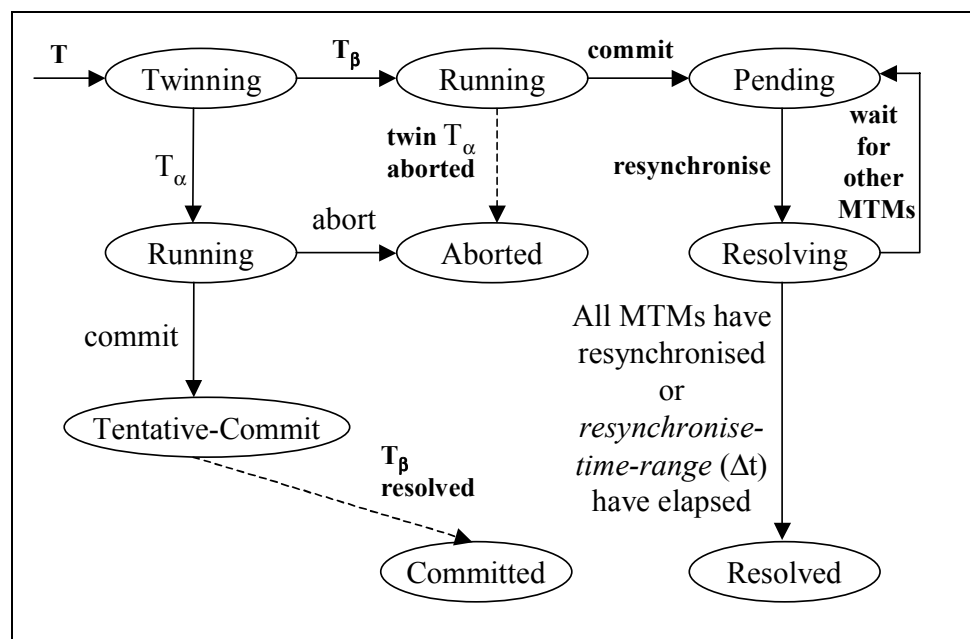


Figure 4.2 Twin-Transaction states in connected mode

Any subsequent transactions that operate on results produced by a pending transaction T will become dependent on *success* of transaction T . There can be multiple pending transactions creating possible dependencies among themselves. This inter-dependency of transactions (twin-transactions) must be captured to eliminate any inconsistent operations/transactions. This will include *read* and *write* dependencies. The *read* dependency occurs when a transaction T' reads a data item written by a pending transaction T and *write* dependency occurs when a transaction T' writes a data item written by a pending transaction.

A transaction T reaches its *resolved* state when all the transactions on which transaction T is dependent have reached their resolved states. Since, MTMs are operating in disconnected mode, the transaction T cannot reach its *resolved* state unless all the MTMs have connected with FTM and have gone through the resynchronisation process. This is necessary to ensure that every transaction that executed during disconnection has been taken into account. But, it is possible that some MTMs might not connect. In such a case the FTM will always be in a *tentative* state and none of the transactions will reach their *resolved* state. A timeout for each MTM is required to avoid waiting for a connection by that MTM indefinitely. If an MTM connects after the timeout value, the transactions executed on that MTM are resynchronised *after* all the *resolved* transactions.

4.2 Transaction History

Each mobile host executes a transaction in both connected and disconnected modes. These transactions need to be synchronised with all the transactions executed by all other hosts to achieve a global consistent state. For that purpose each twin-transaction

manager creates a transaction history log (H). The transaction history log is used to synchronise transactions during the resynchronisation process that propagates the transaction from an MTM to the FTM. The conflict detection (consistency validation checks) cannot be performed if the transaction system does not record history of information about disconnected transaction executions (transaction history). Certain strategies are adopted to keep the size of the log to a minimum. This results in better performance and more of the log is accepted as valid during resynchronisation process.

Each transaction reads and writes data items. The set of data items read by a transaction is called the READSET of that transaction. The set of data items written by a transaction is called the WRITESSET of that transaction.

The transaction history log (H) captures the READSET and WRITESSET of each transaction and builds a transaction *precedence Graph G*. This *precedence graph* is based on a serial history log *H*.

If $H_k = T_{k1}, T_{k2}, \dots, T_{km}$ be the serial history log for a twin-transaction manager TTM_k , then the precedence graph G_k is a directed graph having as nodes transactions executed on TTM_k . The edges of a precedence graph G_k are of the following two types:

- *Dependency Edges*. Let T_{ka} and T_{kb} be two transactions executed by TTM_k with $a < b$ (a executed before b). Then there is a dependency edge $T_{ka} \rightarrow T_{kb}$ in G_k if $WRITESSET(T_{ka})$ and $READSET(T_{kb})$ have nonempty intersection.
- *Precedence Edges*. Let T_{ka} and T_{kb} be two transactions executed by TTM_k with $a < b$ (a executed before b). Then there is a precedence edge

$T_{ka} \rightarrow T_{kb}$ in G_k if $READSET(T_{ka})$ and $WRITESET(T_{kb})$ have nonempty intersection.

A *global precedence graph* G is a directed graph where nodes are transactions on n twin-transaction managers $H = H_1, H_2, \dots, H_n$. The *global precedence graph* G is constructed by combining *precedence graphs* of each of the twin-transaction managers. Apart from the above-mentioned edges, a third kind of edge (called *interference edge*) is constructed during resynchronisation (combining of *precedence graphs*).

- *Interference Edges.* Let T_{ka} and T_{lb} be two transactions executed by TTM_k and TTM_l , respectively. Then there is an interference edge $T_{ka} \rightarrow T_{lb}$ in G if intersection of $READSET(T_{ka})$ and $WRITESET(T_{lb})$ sets is nonempty.

The *dependency* and *precedence* edges are used to detect conflicting transactions within one TTM and the *interference* edges are used to detect conflicts among transactions executed by different TTMs. The following theorem presented in section 2.2.2, [Dav84b] remains true even for twin-transaction model.

Theorem: *The combined histories are serialisable if and only if their precedence graph is acyclic.*

As long as G is acyclic, it describes a partial order on transactions and any serial logs induced by a topological sort of G represents an equivalent serial one-copy log. If G contains a cycle, then transactions must be backed out to make G acyclic. Clearly, whenever a transaction T is rolled back, all other transactions that are connected to T via

a dependency edge must be rolled back as well, even though these were successfully executed by the individual TTMs.

The global precedence graph G is modified each time a TTM connects and goes through the resynchronisation process. A transaction T , executed by a TTM and is resynchronised, will be in a *global tentative state*. This is to ensure that transactions executed on other TTMs (TTM who have also replicated the data items operated upon by transaction T) are not in conflict with T . There is this possibility that a certain TTM might not connect for a long time. To overcome this situation, a *resynchronise-time-range* (Δt) is associated with each TTM. A transaction T executed on a TTM at time t must be resynchronised with FTM in time interval $[t, t + \Delta t]$. If resynchronisation is done after this time, the success probability of transaction will decrease significantly as a transaction T' executed by some other TTM might have changed state from *global tentative* to *resolved* during that time. The *resolved* transaction cannot be aborted/compensated and future transactions must operate on the new versions of data items produced by this transaction.

For example, let us suppose that transaction T_i executed on data items X_a and X_b on MTM_1 at time t_1 and a transaction T_j executed on data items X_a and X_c on MTM_2 at time t_2 (and $t_1 - t_2 < \Delta t$). Now if MTM_2 connects with FTM within time interval $[t_2, t_2 + \Delta t]$ and resynchronise, transaction T_j will reach its *tentative state*. If MTM_1 resynchronise within time interval $[t_1, t_1 + \Delta t]$ then T_i will go through the resynchronisation process with T_j . On the other hand if MTM_1 resynchronise after time interval $[t_1, t_1 + \Delta t]$, the T_i is in *write/write* conflict as it operated on an older version of data item X_a . FTM will *not*

know with which transaction it is in conflict as that transaction (T_j) has already reached its *resolved state* and has been removed from the *global precedence graph* G .

A resynchronisation relation \Leftrightarrow is defined. This relation defines the resynchronisation of serial history logs $H_1, H_2, \dots H_n$ into a common global serial history log H .

$$H = H_1 \Leftrightarrow H_2 \dots \Leftrightarrow H_n$$

Alternatively, this resynchronisation relation \Leftrightarrow can also be defined as resynchronisation of precedence graphs $G_1, G_2, \dots G_n$ into a global precedence graph G .

$$G = G_1 \Leftrightarrow G_2 \dots G_n$$

This common history (H and G) is then propagated to all the twin-transaction managers, thus making all of them in sync with each other. This operation of resynchronisation will thus guarantee consistency.

To guarantee serial execution of transactions on each MTM for local transactions, the precedence graph must be acyclic. Because of 2PL concurrency control (section 3.4.1), the local precedence graph will be acyclic (thus guaranteeing a serial execution of transactions).

The maintenance of precedence graph requires the transaction manager to check every operation of a transaction. This could decrease the performance of the overall system. To avoid this check for every transaction operation, the maintenance of dependency

edges of precedence graph can be partly done by concurrency control mechanism. All that is required:

- Insert a new node into precedence graph each time a transaction starts execution. This can be done during the twinning process (α -twin inserted into precedence graph).
- Create a dependency edge when a transaction needs to wait for another transaction.

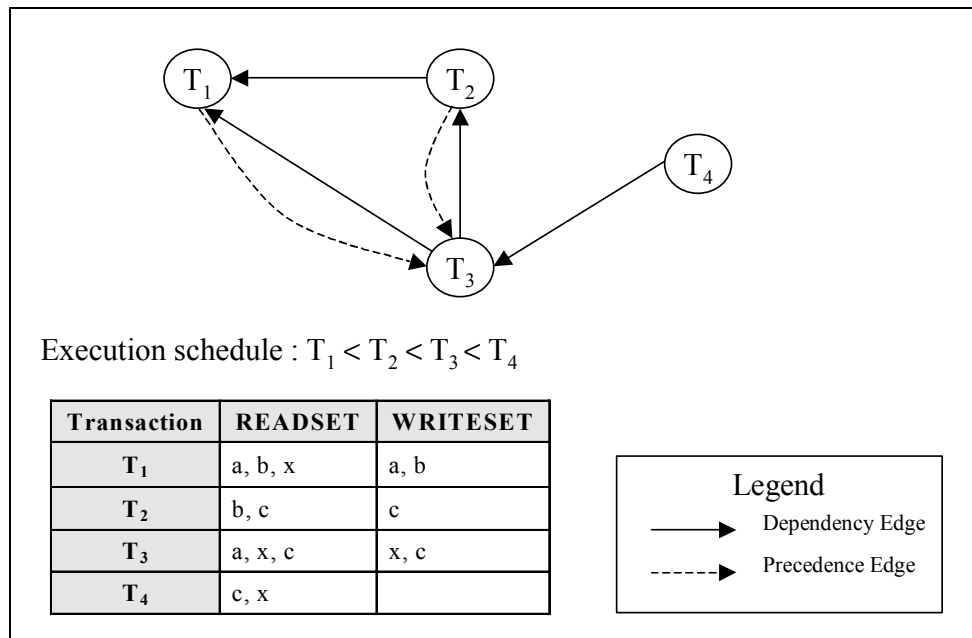


Figure 4.3 Precedence Graph

The rest of the edges (dependency and precedence edges) can be constructed when the transaction commits. This is be done by comparing the *READSET* and *WRITESET* of concluding transaction with all the pending transactions. A sample execution schedule and its precedence graph are shown in Figure 4.3. The Figure 4.3 lists four transactions and their *READSET* and *WRITESET*. The transactions are executed in the order $T_1 < T_2 < T_3 < T_4$. The precedence graph as a result of applying the above-mentioned rules is also presented in the figure.

The *READSET* and *WRITESET* of transactions, that are executed earlier and are still in *pending* state, capture the state of transaction system when a transaction is executed.

4.2.1 Intelligent Log Maintenance

The disconnected operation of the twin-transaction model is costly as it requires persistent storage space for recording the precedence graph for the α -*twin* of transactions and a transaction table for the transactions. Maintenance of precedence graph also requires computing resources.

The resynchronisation costs are also dependent on the size of the precedence graph. The size of the precedence graph also dictates how often an MTM should connect with FTM. Thus, it is necessary for the twin-transaction system to reduce the size of the precedence graph (and thus transaction history log). This can be achieved by removing those transactions from the precedence graph and the transaction table, which no longer have any effect on the overall transaction system.

The detection and removal of a *redundant* pending transaction must preserve certain correctness conditions. These correctness conditions can be general or application specific. In the current implementation of the twin-transaction model, there is no provision for defining an application specific detection of *redundant* pending transactions.

The criteria for detecting a *redundant* transaction is adopted from the IOT model [Lu96] and is defined as:

- **Obsolete.** A transaction is *obsolete* if the net effects created by that transaction are either eliminated or made obsolete by later transactions.
- **Offspring Relation.** Two transactions T_1 and T_2 have an *offspring relation* if T_2 cannot be re-executed if effects of T_1 are removed.
- **Covered.** A transaction is *covered* if the removal of a transaction from the local transaction history will not affect the resynchronisation process of any other pending transaction. This can be guaranteed if the transaction has no *offspring relation* with any other pending transaction.

For example, suppose that a transaction T_a inserted a data item X_a and a later transaction T_b deleted data item X_a . Also suppose that no other transaction has a WRITESET or READSET containing X_a .

- Transaction T_a is obsolete.
- Transaction T_a and transaction T_b have an *offspring relation* (since T_b cannot be re-executed if effects of T_a , ie insertion of data item X_a , are removed. T_b cannot delete X_a).

Thus transaction T_a is not *redundant*. But if transaction T_b does not have any other dependency or precedence edge then it also is obsolete and both these transactions can be removed from the transaction history log as they both are obsolete and they don't have any *offspring relation* with any other pending transaction.

The detection and removal of *redundant* transactions increase the chance for the complete transaction history to pass validation and commit their results. This is due to the fact that it is possible that some of the cancelled transactions may cause conflicts

with some other transactions during the resynchronisation phase. Although detecting a *redundant* transaction is a complex process, the benefits of removal of a *redundant* transaction outweigh the detection cost.

The process of detection of the *redundant*-transactions needs to be executed on the completion of each transaction. It consists of three steps, ie detecting *obsolete* transactions, checking for *offspring* relations and see if the transaction is *covered*.

For each completed transaction T , the transaction manager scans the precedence graph in reverse order. Any operation that becomes *obsolete* due to T is marked. If all the operations of a transaction become obsolete, the transaction becomes obsolete. The transaction manager will also mark the transaction T obsolete if its operations are offsetting operations of other obsolete transactions.

For each *obsolete* transaction T , the transaction manager scans the precedence graph for any transaction that have read a result written by transaction T . If no such transaction is found, the transaction T is marked *covered*. If such a transaction T^0 is found and it is not *obsolete*, then transaction T is not *covered*. On the other hand if transaction T^0 is *obsolete*, then a check is made to see if transaction T^0 is *covered* or not, if *covered* then transaction T is also *covered*.

All the transactions that are marked *obsolete* and *covered* are *redundant* transactions and are removed from the transaction history.

Redundant operations can also occur within a transaction. That is, a transaction may consist of a number of operations and some of those operations cancel each other's effects (for example creation of a temporary record and then removing it). The algorithm defined above for detecting *redundant* transactions can also be applied within the scope of a transaction to detect *redundant* operations and remove them from the transaction log.

Apart from detecting *redundant* transactions, the detection of *obsolete* transactions is also beneficial. The results produced by the *obsolete* transactions are superseded by results that are produced by other transactions. The *obsolete* transactions are kept in the log but their *obsolete* property is used during the resynchronisation process. If a set of transactions that made a transaction *obsolete* is serialisable then the *obsolete* transaction is also serialisable and since its results have already been made *obsolete*, its log entry does not need to be processed (thus minimising the amount of work done during the resynchronisation process). This is particularly useful in cases where a periodic transaction updates a particular data item. If that transaction ran for a number of times, then the resynchronisation process doesn't need to process the log entries for each execution as long as it can successfully process the last one.

4.3 Concurrency Control

The twin-transaction model in essence is an optimistic transaction management model. The transactions executing on different disconnected clients cannot read results produced by other clients (during disconnected or connected modes). This necessitates that even the results produced by the connected clients must also be kept in a pending state (for a specified amount of time, or until all clients have resynchronised).

The twin-transaction model operates in two modes: connected and disconnected. The execution of a twin-transaction in the two modes is different. In connected mode, FTM reflects the effect of a transaction straightaway. To ensure that these effects are consistent with already committed transactions, a concurrency control mechanism is required. This requirement and a solution is discussed in section 4.3.1. A local concurrency control mechanism is also required to ensure that transactions executing locally on an MTM are at-least serialisable locally before being tested on FTM for conflicts.

In disconnected mode, the results of a transaction cannot be transferred to FTM. The transactions that are executed locally are serialised by the local concurrency control mechanism and are placed in a history log. This history log is transferred to FTM during reconnection and the resynchronisation process is executed to achieve a global consistent state. This is discussed in section 4.4.

4.3.1 Connected Mode Operation

The twin-transaction model promises to achieve one-copy serialisable schedule for twin-transactions. In a connected environment, the twin-transaction model has two options while executing a transaction:

1. Place β -twin in pending state and wait for resynchronisation.
2. Submit transaction to FTM for execution and compare results with that of a local execution.

The First option is analogous to disconnected operation and is not adopted in connected mode operation. In second option, the resynchronisation process is avoided in favour of direct execution on FTM. Even though the transaction is executed on FTM, its β -twin will still stay in a *pending* state on FTM (section 3.2.3 and 4.1) and will reach its final state when all other MTMs have resynchronised with FTM.

4.3.1.1 Concurrency Control Design Alternatives

There are a number of concurrency control techniques for enforcing one-copy serialisable schedule among concurrent transactions. The applicability of these techniques for the twin-transaction model is discussed in the following paragraphs.

Lock Based Approach

In this approach locks are used to coordinate accesses on shared objects among concurrent transactions. It is the most commonly used method and is used in a number of commercial database systems. Lock based approach can be used to ensure one-copy serialisable schedule in the twin-transaction model but it is not suited for it. The reasons are:

- Management of distributed locks is difficult and costly
- Unpredictable disconnections
- Standard locking (2PL) is sensitive to long running transactions

Time-Stamp Based Method

In this method each transaction is attached with a unique time stamp and each object accessed by ongoing transactions is also attached with a unique stamp information. Serialisability is achieved by enforcing time stamp order for all conflicting data access

operations that are issued by concurrent transactions [BHG87]. It is a good candidate for twin-transaction model but its main disadvantage is the need for immediately updating time-stamp information of objects, otherwise global synchronisation of concurrent transactions is not possible.

Optimistic Concurrency Control

In optimistic concurrency control (OCC), transaction execution is performed within a private workspace. At the end of the execution, a check is made against the public space to see if the results produced can be serialised with previously committed transactions. If so, the results are committed to public space, otherwise, it is thrown away and the transaction is re-executed [KR81].

4.3.1.2 Selected Concurrency Control Mechanism

The optimistic concurrency control (OCC) mechanism is chosen for the twin-transaction model. There are a number of advantages of using OCC for the twin-transaction model (even for connected mode):

- Better performance. Previous studies indicate that OCC offers strong performance if likelihood of contention is low [DTK88].
- Can be implemented efficiently.
- Network traffic is minimised, as communication bandwidth is a critical resource in mobile computing environment.
- A perfect fit for proposed twin-transaction connected mode operations (section 3.2.3).

The twin-transaction model assumes that β -twin will be successfully executed on FTM and the results produced by both twin transactions (α and β) will be the same. Figure 4.2 shows that the *commit* of β -twin is dependent on that of α -twin and the transaction reaches the *resolved* state after the resynchronisation process. Although, the transaction is executed in connected mode, the resynchronisation process still needs to be executed and the transaction will go through the *pending* state before reaching the *resolved* state. This is due to the fact, that there are transactions that might be executed on other disconnected TTMs that needs to be resynchronised with transactions executed in connected mode.

The advantages of having twins of a transaction in *running* state during connected mode are:

- The communication between the two TTMs is minimised.
- Only the end results are compared and if different the results produced by α -twin are discarded and results produced by β -twin are replicated to the MTM.
- If the MTM that submitted β -twin of a transaction disconnects while in the middle of a transaction, FTM can safely execute β -twin and the result can be resynchronised when that MTM connects again. Thus MTM doesn't need to wait for the completion of a transaction.

Another approach, to execute α and β twins, involves executing α -twin on MTM and not sending β -twin to FTM. The *READSET* and *WRITESET* of α -twin can be sent to FTM for resynchronisation. If the results produced by α -twin can be serialised with previously committed transactions on FTM then β -twin is not sent to FTM at all

(minimised computing load on FTM). On the other hand if results of α -twin are not accepted, then β -twin is executed on FTM and results are propagated back to MTM. This approach will have advantage only if the cost of sending and running β -twin transaction is less than the cost of sending and resynchronising of READSET and WRITESET of α -twin. The disadvantage is that FTM will not be able to place the transaction in *pending* state for other MTMs who are disconnected at that time. This is the main reason why such a scheme is not adopted for the *twin-transaction model*.

The twin-transaction model uses replication to provide a private workspace for the execution of α -twin. The public space is kept on FTM where all β -twins are committed and the results and transaction history logs are maintained. The execution of a transaction and its validation on FTM and MTM is done in two phases.

Phase 1:

- Initially the two twins T_α and T_β are executed on MTM and FTM respectively.
- Transaction T_β on FTM goes through the *twining process*. It reaches *pending* state when its α -twin is in a *tentative-commit* state. The results produced by its α -twin are checked against the public space for conflicts.
- The results produced by transaction T_β in *pending* state are sent back to MTM.
- MTM checks the results against that of T_α . If conflicts are detected, the results produced by T_α are discarded and received results are *tentatively* committed.

- If T_α is aborted on MTM, T_β is also aborted on FTM.

Once the transaction is validated by OCC in phase 1, the next phase of validation begins that is required by the twin-transaction model.

Phase 2:

- The results committed on FTM can only be finalised when all other disconnected MTMs have resynchronised with FTM or *resynchronise-time-range* (Δt) for all TTMs have elapsed.
- Due to this reason, the results that are committed by MTM, even in connected mode, are kept in a pending state.

When a transaction fails validation during resynchronisation, although it was committed and was validated by OCC on FTM (during connected mode), the class of transaction decides the outcome of such failures. This situation is triggered by the resynchronisation process of MTMs that are connecting with the FTM. It is discussed in detail in section 4.4 and chapter 5.

4.3.2 Local Concurrency Control

The optimistic concurrency control mechanism discussed in section 4.3.1.2 only takes care of concurrency among transactions executed by different TTMs. A second level of concurrency control mechanism is required to enforce local consistency model (section 3.4.1). This is required to make sure that concurrent execution of *α -twins* of transactions on a TTM is one-copy serialisable. These transactions must be synchronised so that the replicated copies of data items remain consistent for each of them.

In current implementation strict two-phase locking (2PL) protocol is used for local concurrency control. The advantages of 2PL are:

- Simplicity
- Reasonable performance (when data sharing is infrequent)

Since a single user typically operates each MTM, the likelihood of concurrent transactions operating on the same data (read/write sharing) is low. Also 2PL is simple to implement.

Deadlock

The disadvantage of 2PL protocol is that it is possible for a group of transactions executing on the same MTM (local concurrency control) to deadlock. That is, when a transaction T_1 need to wait for another transaction T_2 due to 2PL and transaction T_2 need to wait for transaction T_1 . The wait might be direct or indirect (T_1 waiting for T_{11} waiting for ... T_2 waiting for T_1).

In current implementation of the twin-transaction model, the standard deadlock detection technique of maintaining a *wait-for-graph* (WFG) is used. Each node of WFG represents a running transaction T denoted by $wfg(T)$. When a transaction T needs to wait for another transaction T' , an edge from $wfg(T')$ to $wfg(T)$ is inserted into WFG. Deadlock is detected by checking for cycles in WFG. If a cycle is found, it means that the corresponding transactions on the cycle are deadlocked. When that happens, the transaction system will print out messages notifying the users that this group of transactions are deadlocked and some (or all) of them must be killed in order to make progress. The user must explicitly kill transactions. It is not done automatically.

4.4 Disconnected Mode Operation

The twin-transaction system running on an MTM that is disconnected has a number of responsibilities. The results produced by the local transactions are tentative and are dependent on validation by FTM on reconnection. To achieve a consistent state with FTM it needs to perform a number of tasks to make sure that enough information about executed transactions is made available to FTM. These tasks include:

- Maintaining local consistency (section 4.3.2).
- Recording transaction history information.
- Detecting redundant disconnected transactions.
- Probabilistic Success/Failure calculations.

The twin-transaction model ensures the consistency of local state of a disconnected MTM by requiring disconnected transactions to satisfy local serialisability. This is achieved by local concurrency control and is discussed in detail in section 4.3.2. The rest of the tasks are discussed in the following sections.

4.4.1 Probabilistic Conflict Detection

In disconnected mode of operation, the twin-transaction manager has no knowledge of the state of other TTMs. In such a situation an optimistic approach to concurrency control is adopted. The transaction is allowed to execute under the local concurrency control mechanisms and conflict detection is done during the resynchronisation process. To decrease the number of conflicts during the resynchronisation process twin-transaction model proposes a *probabilistic conflict detection mechanism*. The mechanism relies on computing probability of success/failure of a transaction that is being executed in disconnected mode.

The calculation of probability of success/failure for a transaction is called *conflict-estimation* and it is done by the *conflict-estimation-agent*, which is part of the twin-transaction model. The twin-transaction implementation should provide a default *conflict-estimation-agent* and optionally should have the provision for the programmers to define their own *conflict-estimation-agent* for a class of transactions.

By determining the success/failure rate for a class of transactions, the twin-transaction managers ensure that possible conflicts arising from such transactions are avoided in the first place. In current implementation of twin-transaction model the default *conflict-estimation-agent* achieves this by following one of the following control paths:

- Not allowing such transactions to execute.
- Placing transaction in the log for later execution (on FTM).
- Informing *users* of possible failure and asking for confirmation.

The calculation of probability of success/failure of transactions is dependent on a number of factors. These factors are implementation and application specific but in general terms they can consist of any of the following data elements.

Time of execution of a transaction.

The time of execution of a transaction can also have an impact on its chances of succeeding during the resynchronisation process. For example, in a business organisation most of the work is done during the daytime and therefore any mobile host that is disconnected runs into a higher chance of causing conflicts. Whereas during

nighttime, it is less likely that the transactions that are being executed in disconnected mode will cause conflicts (if mobile host resynchronise it self before office hours).

Number of transactions that are already in pending state.

The transactions cause conflicts if they operate on same data on different hosts. In the twin-transaction model, the transactions are kept in pending state until they are resolved. If the number of transactions in *pending* state ($N_{pending}$) increases, it means the data set on which those transactions operated will also possibly increase. This in turn increases the probability that some mobile host will execute transactions that can possibly access a data item that has already been accessed by a pending transaction.

Dependency and Precedence edges on pending transactions G^d .

The transactions are kept in *pending* state until they are resolved. In the meantime more transactions are being executed that depends/precedes the results produced by the *pending* transactions. A transaction that depends/precedes more *pending* transactions, has a higher chance of failure during resynchronisation (as failure of any one of those *pending* transactions can cause failure for this transaction).

Possible interference edges on pending transactions

Interference edges are created during the resynchronisation process of transactions executed by the twin-transaction managers. The calculation of possible interference edges in disconnected mode is complex. This can be defined as a probability for a class of transactions (or transaction - data item combinations). The twin-transaction manager also has an agent that calculates this probability based on previous *interference* edges that resulted during the resynchronisation process (chapter 5).

DEFINITION:

A transaction T in history log H_k of TTM_k is in conflict with a probability ρ with some transaction that might be executed by another TTM . Where ρ is a function of a number of factors that are implementation and application specific. Some of these are described above.

$$\rho = F(T, t, N_{pending}, G^d, G^p)$$

For each class of transaction the probabilistic conflict detection function (ρ) determines whether the transaction should be allowed to proceed or not. In case of a probabilistic conflict the required action is also defined by the function.

This probabilistic conflict detection (PCD) is performed when a transaction concludes. It is possible for a *user* application to ask explicitly for the PCD value of a transaction and aborts itself if it is too high.

It is always difficult for the transaction programmers to determine the criteria under which a transaction might get aborted on resynchronisation. Therefore the definition of a conflict detection algorithm for a class of transactions is hard. The twin-transaction model also proposes an automatic probabilistic conflict detection (APCD) process. That liberates the transaction programmers from defining a conflict detection algorithm.

The APCD process relies on automatically learning conditions and states under which the chances of a conflict are higher. A specialised agent, *pattern recogniser and learner*

(PRL), detects and learns these conditions and states. The learned conditions and states are called a *pattern* for a class of transactions under which the probability of conflict is higher. These patterns are stored in a database and are used by APCD process. A particular implementation of the twin-transaction model can implement a tool to populate this *pattern database* with some initial patterns that are later on modified by PRL (in the due course of time). A particular implementation of the twin-transaction model might not implement PRL or disable PRL for a class of transactions (made it static rather than dynamic) and can simply rely on this tool (and thus on user) to populate/modify *pattern database*.

The twin-transaction behaviour is controlled by the knowledge (rules, patterns) stored in the *pattern database*. Its up to users of the twin-transaction model to make sure that the knowledge defined is a *correct* estimation of transaction behaviours.

During the resynchronisation process *the pattern recogniser and learner* is informed of the result of the resynchronisation process. The PRL learns from the result to allow/restrict transactions that might result in more work during the resynchronisation process.

4.4.2 Transaction Execution Rules and Patterns

The twin-transaction model allows transactions to execute on different TTMs and any conflicts arising due to independent execution of these transactions are detected and resolved during the resynchronisation process. To decrease the number of conflicts (and amount of work required to redo/eliminate the transaction) the twin-transaction model relies on the calculation of probability of success/failure of a transaction. The

calculation of conflict-probability ρ of a transaction is dependent on a number of factors (as described in section 4.4.1). These factors are used to predict the environment under which a transaction's conflict-probability will rise. For that reason, the twin-transaction model adopts a logging/learning mechanism that keeps the information about a transaction's resynchronisation result and under what environment that result was achieved. This knowledge about the possible outcome of a transaction under a particular environment is used to calculate conflict-probability ρ for similar future transactions.

4.4.2.1 Transaction Execution Pattern (π)

The transaction, its resynchronisation result and the environment under which that result is achieved is called a *transaction-execution-pattern* (π). The *transaction-execution-pattern* for a transaction is captured by the *pattern-recogniser-and-learner* during conflict-estimation and the resynchronisation process. A database of π is also kept to help calculate the conflict-probability of *similar* transactions. The determination of *similarity* of transactions is discussed in next section.

The environment of a transaction can be composed of a number of elements (section 4.4.1). Although it is tempting to use all these elements to record the execution pattern of a transaction, it is not practical. The database of transaction execution patterns will be too big and search on a big database will also be slow. In current implementation, the transaction execution pattern of a transaction is defined in terms of data items that have been accessed during and before the execution of a transaction.

$$\pi(T) = \{X_{1}^{op}, X_{2}^{op}, \dots, X_{n}^{op}\}$$

Where $\pi(T)$ is the transaction execution pattern for transaction T and X is the data item that fulfils one of the following conditions:

- Transaction T accessed it.
- It is a member of $\pi(T')$, such that transaction T accessed a data item Y that was also accessed by a transaction T' .

The data access operation performed on data item X and how that data item is added to $\pi(T)$ (one of the above conditions) are represented by attribute op of X .

The size of $\pi(T)$ is dependent on how many transactions have executed before transaction T and what sort of operations they performed. For every new transaction, the size of $\pi(T)$ will increase as the numbers of transactions that have already executed increased.

4.4.2.2 Transaction Pattern (π -) Database

The twin-transaction manager captures the transaction execution pattern and during the resynchronisation process that pattern is used to determine the success/failure of that transaction. To enhance the probability of success during the resynchronisation process, the twin-transaction model adopts a logging/learning mechanism that keeps the information about transaction execution pattern and its resynchronisation result (under a particular environment). A database of transaction execution pattern and its success/failure ratio is maintained (called π -database). This information is used later on during execution of a *similar* transaction (section 4.4.2.3) to determine the *conflict-probability* ρ of that transaction.

The π -database is updated when a transaction is executed and resolved. The pattern database stores transaction execution pattern as two entities:

- Transaction execution environment
- Transaction pattern

The *Transaction pattern* corresponds to data items accessed by a transaction T where as the *transaction execution environment* consists of all other data items that are added to $\pi(T)$.

These elements are defined in terms of *internal* patterns. Rather than storing a complete set describing one of the above entities, each entity is sub-divided into multiple sub-entities (sub-patterns). These sub-patterns are stored as *internal* patterns. This division is used to minimise the size of pattern database. One extra attribute is stored with each *internal* pattern describing whether it represents the start of a transaction pattern or not.

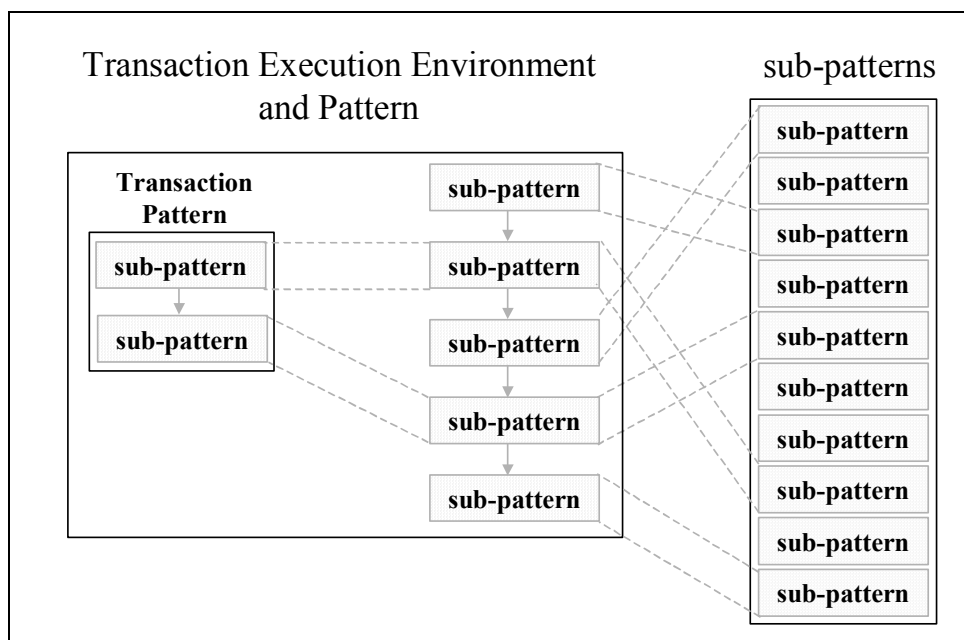


Figure 4.4 Architecture of π -Database

In Figure 4.4, the *internal* patterns are shown as a set of *sub-patterns* and the structure of a transaction pattern and its environment is shown in terms of these *sub-patterns*.

4.4.2.3 Transaction Similarity

The conflict estimation portion of the twin-transaction model calculates the conflict-probability ρ of the transactions. During the execution of a transaction, the pattern-recogniser-and-learner uses transaction execution pattern to determine which $\pi(T)$ in π -*database* matches. This is done by the following steps:

- The transaction pattern is matched against transaction patterns in π -*database*.
- The transaction execution environment is matched against the transaction execution environments of matched transaction patterns.

This two-step algorithm ensures that not too much time is wasted in calculating success/failure probability.

The *similarity* (θ) of transactions is defined as a percentage based on two kinds of matches:

- Same set of data items is accessed.
- Data items are accessed in the same sequence.

The similarity of transactions increases if the same kind of operation is performed on the same data item.

A Transaction might not access all data items in a sub-pattern. The transaction will still be considered compatible to such pattern if it operates on at least a minimal set of objects. This minimal set can be defined with the sub-pattern using the *rule-definition tool*.

4.4.2.4 Conflict-Estimation

Once it is determined that a particular transaction is *similar* to a set of transactions, then the next step is to determine how closely the transaction execution pattern of the transaction matches with the transaction execution patterns of *similar* transactions. The *similarity* ϕ of transaction execution environment of *similar* transactions is also calculated in the same way, as the *similarity* of transactions is determined.

The last step in conflict-estimation is to calculate the conflict/success probability. This step involves taking into account the following data elements:

- Similarity θ of each *similar* transaction.
- *Transaction execution environment similarity* ϕ of *similar* transactions.
- Failure ratio ω of each similar transaction (how many such transactions had been executed and how many failed during the resynchronisation process).

The conflict probability is estimated using the following formula:

$$\rho = \frac{\sum_{i=1}^n ((\theta_i + \phi_i) \oplus 2) \times \omega_i}{n}$$

Where n is the total number of similar transactions. The three data elements θ , ϕ and ω satisfy the following condition:

$$0 \leq (\theta, \phi, \omega) \leq 1$$

When the similarity θ of all similar transactions is 1 (they are a perfect match) and the transaction execution environment similarity ϕ is also 1 (the execution environment is the same), then the conflict probability ρ is inversely proportional to ω (if ω is 1, then ρ is 0, if ω is 0 then ρ is 1).

On the other hand if similarity θ of all transactions in the pattern database is 0 (ie no similar transaction has been encountered), the transaction execution environment similarity ϕ will automatically becomes 0 (as there are no *similar* transactions). In such a case the conflict probability ρ cannot be calculated and is reported as 0 (optimistic approach – no conflict is suspected).

4.5 Summary

This chapter focused on the execution of transactions in connected and disconnected modes. Each transaction that has been executed goes through a number of states before it is committed. Transaction history log is used to store the transactions that have been executed and are in a *pending* state. For local concurrency control 2PL mechanism is employed. The global consistency is based on optimistic concurrency control mechanism and the resynchronisation process. The resynchronisation process is discussed in detail in the next chapter. The use of transaction execution patterns to determine the probability of success/failure for a transaction is also explained.

Chapter 5

Resynchronisation

In previous two chapters, local/server concurrency control, probability of success and execution of transaction of twin-transaction model are discussed. In this chapter the resynchronisation process that is carried out during the reconnection of a mobile host is discussed in detail. The resynchronisation process is dependent on the transaction history log. The algorithms that are used to perform the resynchronisation process are presented. The data item states during transaction execution and resynchronisation are also explained.

5.1 TTM State Propagation

The resynchronisation process is executed whenever a *Mobile Transaction Manager* (MTM) connects with *Fixed Transaction Manager* (FTM) and passes its *pending* transactions and their execution history log to FTM. The resynchronisation process is executed even though the transaction history log is empty. This is due to the fact that local state of MTM and local state of FTM evolve along their own courses (different transactions take them to different states from an initial state synchronised at last connection). On re-connection, the resynchronisation process will make sure that replicated data items on MTM reflect the updates done on FTM.

The twin-transaction model on an MTM works in three different states as shown in Figure 5.1:

- Connected
- Disconnected
- Resynchronisation

The operation of the twin-transaction model and execution of transactions in *connected* and *disconnected* states is discussed in sections 4.3.1 and 4.4 respectively. The correctness of transactions executed in these two states is tested during the *resynchronisation-state*, when the resynchronisation process is carried out. This is to ensure a *global-one-copy-serialisable* execution.

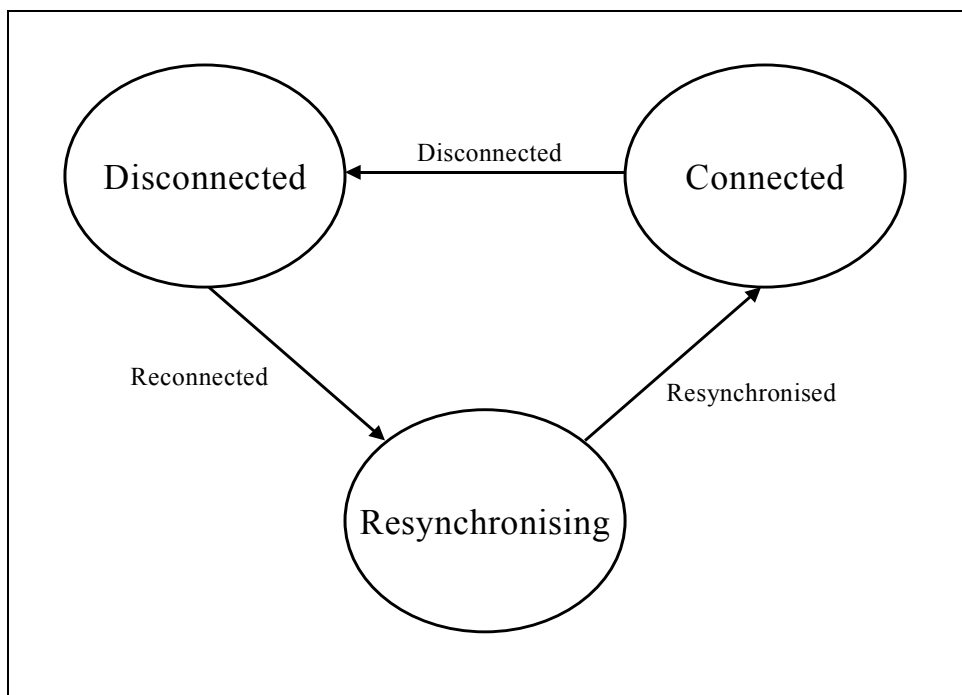


Figure 5.1 MTM states

On reconnection, the MTM goes through the resynchronisation-state before it can start working in connected state. The time an MTM stays in resynchronisation-state before moving to connected-state is dependent on the time required to execute the

resynchronisation process. The time required to execute the resynchronisation process can be arbitrarily long and is dependent on the size of transaction history log on both MTM and FTM. Also if some transactions are detected to be in conflict and require human intervention, the transaction manager has no control over how long it will take. Thus an MTM can remain unsynchronised for a long period of time even though it is physically connected. Thus twin-transaction manager will not be able to transit from *synchronising-state* to *connected-state*.

In *synchronising-state*, new transactions can be executed but these transactions will be executed in mixed mode (connected/disconnected). If the new transaction is dependent on any of the unresolved transactions, then the transaction is executed in disconnected mode and it will also become a pending transaction to be resolved on the server. On the other hand if it is not dependent on any of the unresolved transactions, then it is executed in connected mode.

The propagation of TTM state during the resynchronisation process can be divided into following main tasks and are discussed in the following sections:

- Propagation of *resolved* transactions from FTM to MTM to transit transactions from *tentative-commit* to *commit* state.
- Propagation of *pending* transactions from MTM to FTM for resolution.

5.1.1 FTM to MTM

While an MTM is disconnected from FTM, the state of FTM can change due to the following two factors:

- Other MTMs connect with FTM and resynchronise themselves

- Connected MTMs execute transactions

In both these cases, some transactions will transit from *pending* to *resolved* state. Some of these transactions might update those data items that are replicated on the disconnected MTM. Due to that reason, when an MTM connects and goes through the resynchronisation process it is imperative that the state of MTM should also be checked against FTM. This is to see if any data item is no longer valid and also to check whether some *pending* transactions on MTM updated any invalid data item. In such a case, the *pending* transaction (and its siblings) must go through the resynchronisation phase.

The validation of state of MTM against that of FTM is done in two phases. In the first phase all the data items that are not accessed by transactions on MTM are regarded as immediately resynchronised with the FTM upon re-connection. Those data items that are accessed by the transactions on MTM are considered re-synchronised only after all the transactions have resynchronised on FTM (resulting in either *pending* or *resolved* state on FTM).

For this validation, each data item is marked with a *version* (that can contain Twin-Transaction Manager Identification that last updated it). The version is changed on each update operation for that data item. On re-connection, the resynchronisation process will compare the versions of every replicated data item. If both versions are identical, the data item is marked as valid again, otherwise it is marked as invalid. Any subsequent access to that data item, while the MTM is connected, will cause the twin-transaction manager to re-fetch the new version of data item from FTM. While the MTM is connected, this validation of state of MTM is carried out periodically to keep the status

of data items up-to-date. Before disconnection or on user demand, the twin-transaction implementation should provide a mechanism to re-fetch all invalidated data items.

5.1.2 MTM to FTM

On re-connection, apart from the need to resynchronise data items that have changed on FTM, the transactions executed on MTM are also propagated to FTM (and their results to other MTMs). This is done in the second phase of resynchronisation process when transaction log of MTM is checked against that of FTM. During this resynchronisation process, conflicts are detected and resolved.

The twin-transaction model guarantees *global-one-copy-serialisable* execution schedule. This guarantee requires programmers to deal with conflicts by either programming *application-specific-resolvers* or by manually repairing invalidated transactions. The programmers are also burdened with application specific *conflict-estimation-agents* (section 4.2) that are required to calculate probability of success/failure in disconnected mode operation under certain circumstances.

Since application specific resolvers and *conflict-estimation-agents* work on a transaction-by-transaction base, it is wise to carry out the resynchronisation process of MTM transaction log with that of FTM in the same way. That way, the inconsistency and conflict scope is minimised and the resolver has just one invalidated transaction to work with at a time. Also, the resolver can concentrate on what the transaction has done on MTM and what have changed on FTM during disconnection and make a decision without worrying about interference from other transactions.

The resynchronisation algorithm for a transaction that is in conflict consists of two steps:

1. Invoke the resolver, either application specific or automatic re-execution or abort (if no other resolver exists).
2. The successful resolution of a transaction often requires adjusting the state of those transactions that read from the resolved transaction (read data written by resolved transaction).

The algorithm for the resynchronisation process is given in Figure 5.2.

```

while precedence graph  $G_{MTM}$  is not empty
{
  create a transaction table  $TT[1..n]$  of all transaction in  $G_{MTM}$ 
  for each  $T$  in  $TT$ 
  {
    check for predecessors of  $T$ .  $T$  has predecessor if it has a Dependency or Precedence edge in  $G_{MTM}$ .
    if no predecessor of  $T$  are found
    {
      transit  $T$  into resolving state
      validate  $T$ 
      if validation fails
      {
        determine whether  $T$  can be automatically resolved and if so then
        determine and invoke the resolver
        if  $T$  is in resolving state then request for manual repair
      }
      remove  $T$  from  $G_{MTM}$ 
      place  $T$  in pending state in  $G_{FTM}$ 
      adjust transaction states that were dependent on transaction  $T$ 
      log synchronisation result
    }
  }
}

```

Figure 5.2 Resynchronisation Algorithm

On FTM transactions reach *resolved* state when all MTMs have resynchronised or time-range Δt (section 4.2) have elapsed. After the synchronisation process of each MTM,

the FTM has the option to perform a check on all those transactions that are waiting (to transit to *resolved state*) for this (last) MTM to connect and perform resynchronisation. The transactions that were in such a state transit to *resolved state*.

5.2 Validation Process

Among all the different steps in performing resynchronisation, the validation process of a transaction is the most frequent one. This validation is performed for each transaction against the global state that consists of data items and G_{FTM} . In section 3.4.2 two consistency models (G1SR and GC) are discussed that can be used with the twin-transaction model.

During the validation process, the GC criterion is checked and if a transaction passes the GC criterion then it passes the G1SR criterion as well and is flagged as valid. If a transaction fails GC criterion then if it explicitly requires that GC criterion must be met, then transaction is flagged as not valid, otherwise its validity depends on G1SR consistency criterion.

GC criterion dictates that each transaction must be *serialisable* after all the committed transactions on FTM. The process of verification of a transaction under GC criterion consists of following steps:

- Get *READSET* and *WRITESET* of transaction
- For each data item in these two sets determine whether it was updated on FTM. If updated then transaction fails GC criterion.

In the twin-transaction model, each data item is marked with a version that changes whenever it gets updated (section 5.1.1). That version is replicated along with data item. During GC criterion validation the MTM sends its version to FTM for each data item in the READSET and WRITESSET.

The versions are required to compare two copies (one on MTM and the other on FTM) of a data item. The second option is to allow data items themselves to compare the two copies. The later option is applicable to situations where data item is an object in an object-oriented environment having a compare method.

For G1SR validation, the *READSET* and *WRITESSET* of a transaction is checked against *pending* transactions on FTM. During this process *interference edges* (section 4.2) are constructed for the transaction. There are two cases to be considered:

- If no interference edges are constructed, then the transaction is valid.
- If there is an interference edge $T' \rightarrow T$ and there is no interference edge $T \rightarrow T'$ then transaction T is valid and must be serialised after transaction T' . If there is an interference edge $T \rightarrow T'$ then data items read by T are updated on FTM. The transaction T is invalid (transaction read old values of data items).

5.3 Data Item States

Due to disconnections, the transactions are kept in *pending state* on MTMs and FTM. When a *pending* transaction is invalidated that result in invalidating all those transactions that are dependent on results produced by the invalidated transaction. The twin-transaction manager relies on prediction of conflicts using *probabilistic conflict*

detection (PCD) thus making such cascading invalidation of transactions a rare occasion.

During the PCD process, it is imperative that the probabilistic conflict detection function (ρ , section 4.4.1) should be given the information that data items (accessed by a transaction) are updated by another transaction that is still in a *pending* state. To achieve this, a state-based approach is adopted. A data item is in only one state at a time. The visibility of data item is controlled by its state. A data item can be in one of the following states. The allowed state transitions are shown in Figure 5.3.

- Consistent
- Clean
- Dirty
- Tentative
- Inconsistent

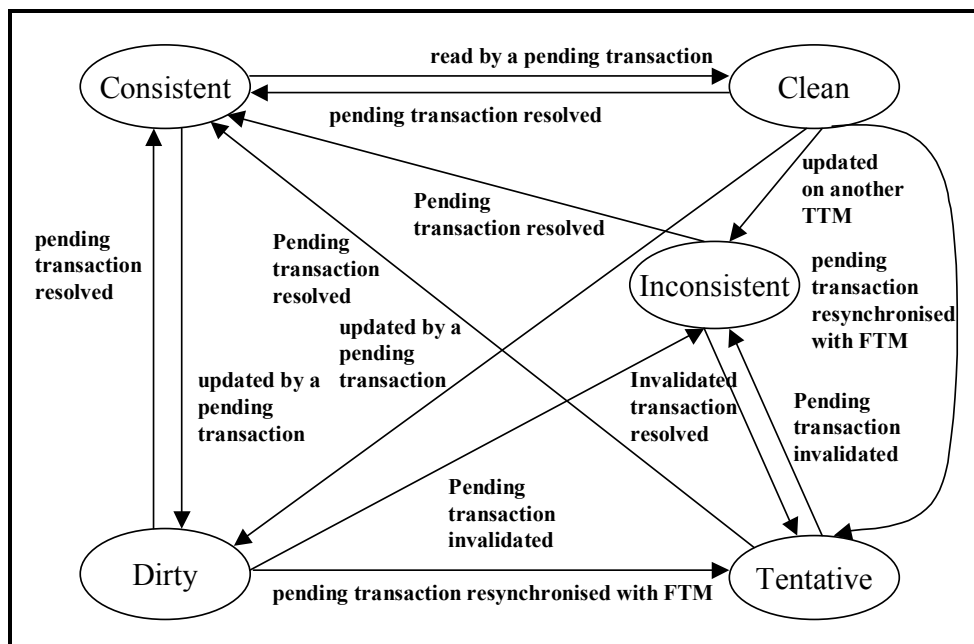


Figure 5.3 Data item states and their allowed transitions

Consistent State

A data item is in *consistent* state if it has not been accessed by any *pending* transaction.

Clean State

A data item is in *clean* state if it has been accessed by a *pending* transaction but is not updated by any *pending* transaction.

Dirty State

A data item is in *dirty* state if it has been updated by a *pending* transaction.

Tentative State

A data item is in *tentative* state if it was updated by a *pending* transaction and has been resynchronised with FTM and is ready to move to *consistent* state when one of the following events occur:

All MTMs have resynchronised with FTM.

Resynchronise-time-range (Δt) has elapsed for all those MTMs which have not resynchronised with FTM.

Inconsistent State

A data item transit to inconsistent state if it has been accessed by an invalidated transaction. The data item stays in this state until the invalidated transaction either reaches *resolved* or *pending* state.

The twin-transaction model exposes *tentative* results of a transaction (data items in *dirty* or *tentative*) state. The PCD process can control the visibility of data items based on the state of TTM and the state of data item. For example, the PCD process can report a high probability of conflict for those data items that are in *dirty* state, thus restricting access/visibility.

Consistent Data Item

A *consistent* data item becomes a *clean* data item when it is read by a transaction. Similarly, when it is updated by a transaction it becomes a *dirty* data item. A *consistent* data item cannot transit to any other state unless it has been accessed by a transaction.

Clean Data Item

A *clean* data item can become *consistent* again when the transaction that read it is resolved. It can also become a *dirty* data item if a transaction updates it. It will become *tentative* data item when the transactions that accessed it are resynchronised with FTM. A worse case is that the read operation of the transaction (that transit the data item to *clean* state) has been invalidated due to an update of the same data item on an other TTM.

Dirty Data Item

A *dirty* data item can either directly becomes a consistent data item or it can become a *tentative* data item. It could also become an *inconsistent* data item when the transaction that updated it has been invalidated during the resynchronisation process.

Tentative Data Item

A *tentative* data item can become *inconsistent* data item if the transaction has been invalidated. In best case scenario, it will become a *consistent* data item.

Inconsistent Data Item

An inconsistent data item can become a *consistent* data item when all the MTMs have either resynchronised or the *resynchronise-time-range* (Δt) have elapsed, otherwise it will become a *tentative* data item.

The twin-transaction model exposes data items in all states (except for *inconsistent* state). The *probabilistic conflict detection agent* (PCDA) is supplied with the state information of each data item and it is up to PCDA to estimate the conflict and avoid transit of data items into *inconsistent* state during resynchronisation.

5.4 Transaction Resolution

When a conflict is detected and a transaction is invalidated, a resolver is invoked to resolve the conflict caused by the transaction. The resolver requires the following basic information about the conflict before it can start to resolve the transactions:

- The *interference edges* that caused the conflict. These edges will give the information that what has changed on FTM.
- The execution schedule of transactions that have operated on data item replicas on MTM and FTM (including the transactions that were executed on other MTMs and have resynchronised with FTM).

Resolution of a conflict caused by a transaction is best resolved when semantics of the application associated with a transaction (that is invalidated) are known. Application specific resolution actions must be supplied by developers of the application (either as a pre-programmed resolver or as a human user). The twin-transaction model supplies a basic framework to perform those actions that do not require application specific knowledge.

The task of transaction resolution can be divided into sub-tasks. The twin-transaction manager performs those sub-tasks that do not require any knowledge of application semantics while remaining sub-tasks are accomplished by application specific resolver. A flow-chart of execution of these sub-tasks is shown in Figure 5.4.

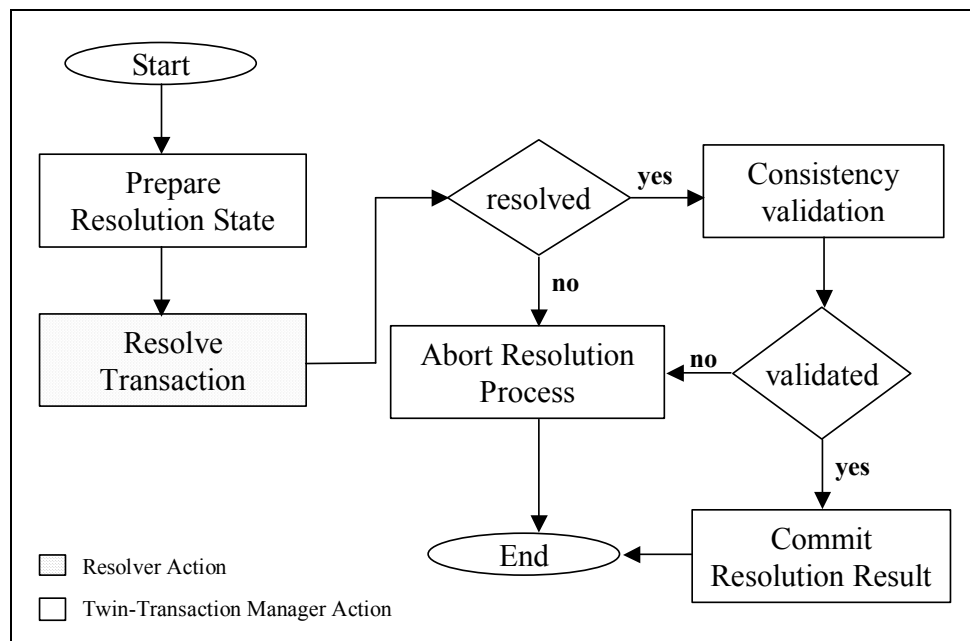


Figure 5.4 Transaction Resolution Process

The transaction resolution process appears as one transaction to twin-transaction manager. It has a start and an end (*commit* or *abort*). There are two advantages of making the transaction resolution process a transaction:

- Any data item access (read/write) is encapsulated within the *transaction-resolution* transaction and is aborted/committed in the same way as any other transaction.
- Intermediate results produced by the resolver are not visible to any other transaction.
- The built-in concurrency control and validation mechanism can be used to validate the result produced by the transaction resolution process (consistency of the result is guaranteed).

The resolution process goes through the steps/actions shown in Figure 5.4. These are explained below:

Start

This marks the start of transaction resolution process (and start of *transaction-resolution* transaction).

Prepare Resolution State

Before the resolver can be started, twin-transaction manager must obtain the information that is required by the resolver to resolve the transaction. This information is captured during this step and is passed on to resolver.

Resolve Transaction

During this process, the resolver is invoked which in turn inspects the state of data items and the transactions that are causing the conflicts and will detect whether the conflict is genuine by utilising application semantics and

will try to resolve it. The outcome of this process determines whether the resolution result is committed or not.

Consistency Validation

The twin-transaction manager will perform consistency checking before committing the results of resolved transaction.

Abort Resolution Result

If the resolver fails to resolve the transaction or the twin-transaction manager fails to commit results of the resolved transaction, then a clean up is performed to make sure that effects of resolver actions are removed from the system.

Commit Resolution Result

On successful validation of resolution results, the results are committed and any clean up required is performed. This clean up includes replacing the data item replica on the MTM with the new value created on FTM.

In section 3.5, five conflict resolution methods are discussed. The transaction resolution process can easily accommodate all five-resolution methods.

Re-execution

The resolver is a transaction on its own.

Abort

The resolver is the twin-transaction manager, which discards the effects of a transaction on MTM. It adjusts the *precedence graph* on MTM to reflect the aborted transaction.

Application Specific

Application specific resolver the most powerful resolver and is supplied by the user (application programmer/developer). It typically checks the data items and the transactions to determine the nature of conflict and then resolve them. The success or failure of resolution is reported by asking for *commit* or *abort* service of the twin-transaction model.

Human Intervention

The twin-transaction model executes a command mode transaction-resolution tool. The user can repair the conflicts by checking data items and transaction states and directly operating on relevant data items.

No-Conflict

In this particular case, the twin-transaction manager will simply either discard the transaction or change the flag of transaction to be valid (thus copying its results to FTM).

When a transaction cannot be resolved using any resolver, the *abort* resolver is used to remove the effects of that transaction. It is imperative that such drastic actions should be logged for later viewing and users should be informed. The twin-transaction manager

logs resynchronisation process in resynchronisation log of each MTM. The resynchronisation log contains entries for each transaction and its resynchronisation history with other MTMs and the resolver that is used to resolve any conflicts. That way user has the option to compensate the effects of a transaction if the result produced by the resolver is not acceptable.

5.4.1 Transaction Resolution Site

For automatic conflict resolution it is imperative that, before execution of any conflict resolver, FTM and MTM must agree upon the site (FTM or MTM) where the conflict resolver will be executed. Obviously, there are only two choices, MTM or FTM. There are a number of issues related to this decision:

- **Security.** FTM (server) might not allow arbitrary resolvers to be executed on a server machine.
- **Scalability.** Resolver execution on FTM costs resources. System scalability can be preserved if resolver execution is shifted to MTMs (client machines).
- **Cost Analysis.** The validation of a transaction is performed on FTM. When a conflict arises, the resource cost of sending the conflict information to MTM must be less than the resource cost of executing the conflict resolver on FTM. This cost analysis will also consume FTM resources and must be taken into account.

Although, the security and scalability constraints suggest that the transaction conflict resolver be executed on MTM, there is one important aspect of the twin-transaction model that forces conflict resolver execution on FTM. The twin-transaction model

keeps the transaction in *pending* state even when the conflict is resolved. The transaction will be considered *resolved* only when it has been resolved against all MTMs. Thus a conflict resolver might get executed multiple times.

5.4.2 Safety and Acceptance of Transaction Resolution

Safety is a critical issue during the transaction resolution process. It becomes more of a concern when *application-specific* resolver (ASR) is used to perform the transaction resolution task, as ASR is not a piece of trusted software. It might contain coding mistakes or some malicious user might code a Trojan horse attack in it.

The acceptance of the transaction resolution results by users is not trivial as transaction resolution is performed transparently. As described in section 3.5, the *abort* resolver is the default resolver if the transaction resolution fails. The *abort* resolver removes the effects of conflicting transaction.

Thus it is imperative that actions performed during transaction resolution process should be reported/logged for auditing by user and/or administrator to make sure that the transaction resolution was safe and transaction resolver did not misbehave.

The twin-transaction manager logs actions performed during the resynchronisation process into a resynchronisation log (one for each MTM). The resynchronisation log contains entries for each transaction and its resynchronisation history with other MTMs and the resolver and its actions that resolved conflicts. That way a user/administrator has the option to compensate the effects of a transaction if the result produced by the resolver is not acceptable.

Apart from this logging facility, the twin-transaction model also imposes strict concurrency control on the *transaction-resolution* transaction. While *transaction-resolution* transaction is running any new transaction T_{new} is not allowed to commit until *transaction-resolution* transaction commits. In case of any conflicts with *transaction-resolution* transaction, the transaction T_{new} is rolled back. This is to ensure that transaction resolution process can continue without any intervention from new transactions. The *transaction-resolution* transaction is run with the same privileges as that of the conflicting transaction. That way, the potential security risk is minimised to those areas where the invoker of the conflicting transaction has relevant rights. To make sure that only valid resolvers are invoked, only an administrator can register application-specific-resolvers.

One potential problem in using application-specific-resolvers is that robustness of the system becomes dependent on the robustness of the application-specific-resolver. If the application-specific-resolver is not well written, the robustness of the system will degrade (becomes equal to that of application-specific-resolver). For example, if a certain resolver gets trapped in an endless loop, the whole system too gets trapped and will not provide any services. To overcome such problems, it is wise to place a limit on the time allowed to resolve a transaction. If the resolver takes longer, then it should be aborted and the resolution process should also be aborted. Since each transaction takes different amount of time to execute, a static value for allowed time would not be acceptable. Since the twin-transaction system already knows how long it took to execute α -twin of the conflicting transaction, it can safely assume that the resolver will take approximately the same amount of time to resolve it. In the twin-transaction

implementation this limit is set at two times the time taken to execute α -twin-transaction.

Atomicity of the transaction resolution is guaranteed due to the fact that the whole transaction resolution process is treated as a transaction. In case of a failure or system crash, during the recovery phase, the twin-transaction system will determine that the *transaction-resolution* transaction did not complete and will simply re-execute the transaction resolution process (ie *transaction-resolution* transaction).

5.4.3 Application Specific and Human Intervention

Resolution Issues

Application-specific resolvers play a fundamental role in maintaining the consistency of the twin-transaction system by performing the conflict/transaction resolution. The viability of the application-specific resolution approach is dependent on a number of factors:

- The developer of a resolver must possess intimate knowledge about internal details of an application. The best candidate for this job would be the developer of the application.
- The availability of the source code of an application is essential to guarantee that the application-specific resolver covers all aspects and have full semantic knowledge of the application.

The resynchronisation process synchronises transactions one by one. The resolver is called to resolve a conflict that is caused by a transaction. Thus the scope of the conflict

is minimised. The resolver is confined to determine what caused the conflict and how to resolve it.

Human-intervention (HI) resolver can also be used as a fallback mechanism in place of *abort* resolver. The HI resolver provides a command line interface to users. Apart from viewing the transaction and data items, the users might want to see which transactions are dependent on the conflicting transactions. To achieve this, the resynchronisation process exposes the *precedence graph* to HI resolver. While repairing a transaction, the user might browse many data items for other purposes and might also update many other data items. Due to these reasons, the concurrency control is relaxed (all data items are readable to HI resolver) and consistency validation is also relaxed. Different commands that a *Human-intervention* resolver should incorporate are:

- **Commit**
Sends a commit command to the twin-transaction manager and terminates the HI resolver.
- **Abort**
Sends an abort command to the twin-transaction manager and terminates the HI resolver.
- **Get-Item**
Display the current value, state of item and any pending transactions that operated on that item.
- **Set-Item**
Set a new value for an item.
- **Get-Transaction n**

Display n^{th} transaction in precedence graph and data items that it has operated upon and its dependency and precedence edges.

5.5 Summary

Different aspects of the resynchronisation process are explained. The algorithms for state exchange among TTMs are presented. The validation process of transactions explains how transaction conflicts are detected by constructing the *interference edges* of precedence graph. In the twin-transaction model data items can go through a number of states. These states are also outlined. Once a transaction conflict is detected, that conflict needs to be resolved. The mechanisms used to resolve conflicting transaction are also discussed.

The next chapter discusses the implementation of the twin-transaction model that was carried out to test the model. The implementation specific issues are also discussed in the next chapter.

Chapter 6

Implementation

The previous three chapters have presented a detailed design of the twin-transaction model. Ways to execute transactions in connected and disconnected modes have been discussed. The propagation of results under different circumstances has also been discussed. That includes conflict detection and transaction resolution. This chapter discusses an implementation of the twin-transaction model. Some of the issues related to implementation are also discussed.

First, the overall architecture and its various components are discussed. The interaction between various components is also discussed. Since many of the basic mechanisms are already discussed in previous chapters, the focus is mainly on those mechanisms that are not yet addressed. These include internal transaction representation and performance issues.

6.1 Architecture

Figure 6.1 shows the skeleton of the twin-transaction system architecture. It consists of two major portions:

- TTM, the twin-transaction manager that handles all the transactions.
- A number of agents to perform necessary tasks to keep the system consistent.

Both server and client components of the system use the same architecture. The difference is in the roles that they perform and in the definition of a data item that is contained in the replicated database (see section 3.2.2).

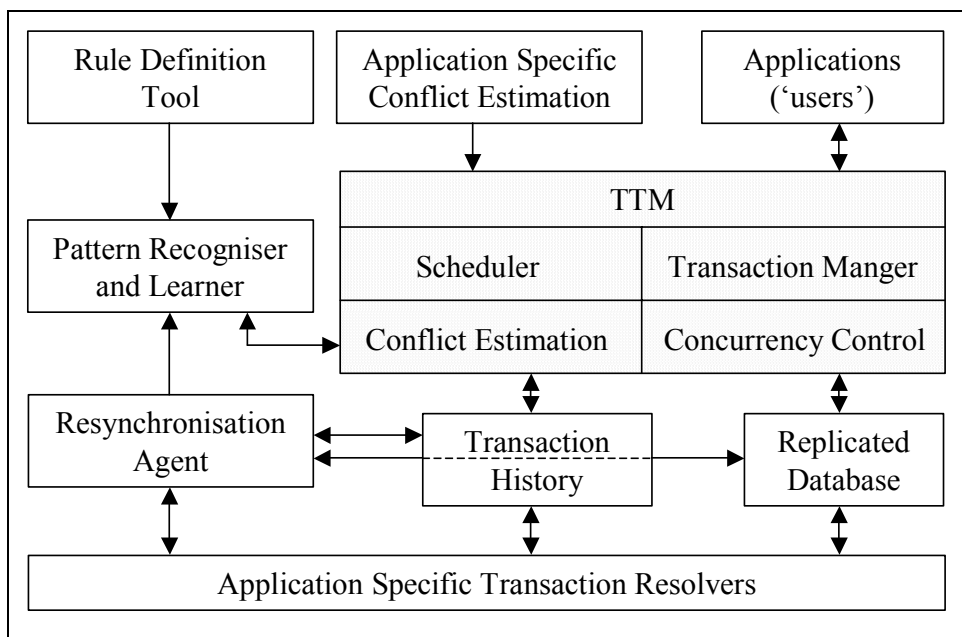


Figure 6.1 Twin-Transaction System Architecture

As shown in Figure 6.1, the transaction system consists of the following parts:

- The twin-transaction manager (TTM), which consists of transaction manager, scheduler, concurrency controller and conflict estimation.
- Replicated database and Transaction history
- Conflict estimation agents and Pattern recogniser and learner
- Resynchronisation Agent and Application specific resolvers

The *transaction manager* is in charge of recording transaction execution information (such as *READSET* and *WRITESET*). It also maintains important data structures (such as *precedence graph*).

The *concurrency controller* performs two types of concurrency control in connected mode, 2PL locally and OCC across MTM and FTM.

The *conflict estimation* is done based on the information available about a transaction. The *rule-definition tool* is used to input transaction patterns for the *pattern-recogniser-and-learner*. The *pattern-recogniser-and-learner* gives heuristic information about the probability of success of a transaction during the resynchronisation process. The transaction patterns and heuristic information gathered about a transaction is handed out to the *application-specific-conflict-estimation* agents who can decide whether a conflict is more probable or not and make a decision on whether to let the transaction proceed or not. It is further discussed in the coming sections.

The resynchronisation agent and application specific resolvers are responsible for performing the vital task of propagating transactions and keeping the whole system in a consistent state.

6.2 Data Structures

The twin-transaction system is a transaction-oriented system and all the modifications to the database are done using transactions. The twin-transaction system records all activities (transactions, sub-transactions) in a transaction database TDB. Its major portions are shown in Figure 6.2, and are explained in the following sub-sections.

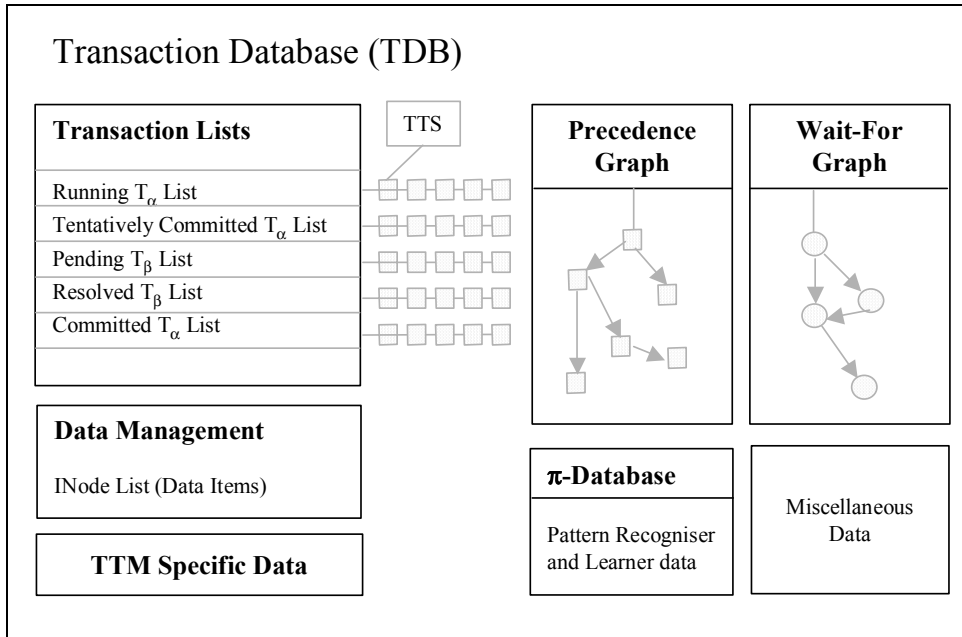


Figure 6.2 Main data structures in Transaction Database

6.2.1 Transaction Lists

The transaction database consists of a number of transaction lists. Each list contains a transaction node (called *t-node*) which represents a particular twin of that transaction in a particular state. For example for a particular transaction T in disconnected mode, the T_α twin will be in *Running- T_α -List* and T_β twin will be in *Pending- T_β -List*. The *Resolved T_β List* and *Running T_β List* only exist on FTM. Each node (*t-node*) in the transaction list contains a pointer to its parent (a twin-transaction structure), which represents the actual twin-transaction. The data structure of the twin-transaction structure (TTS) is shown in Figure 6.3.

By keeping α and β *t-nodes* in different transaction lists (based on their state) improves the performance by reducing the size of a transaction list for a particular state. The reduced size of a transaction list results in efficient frequent internal search activities.

6.2.2 Precedence Graph

The inter-dependency of transactions is captured by the precedence graph (PG). The precedence graph is a mean to capture transaction history on each TTM and is then later used during the resynchronisation process. The calculation of probability of success is also dependent on it. On an FTM, the precedence graph also contains *interference* edges that are constructed during the resynchronisation process.

The nodes of the precedence graph are inserted as transaction activities proceed. The nodes are removed during the resynchronisation process or when a local-transaction is aborted (or its effects are removed). Internally a precedence graph is represented by a set of doubly linked lists.

6.2.3 Wait-for-Graph

On a TTM, during a transaction execution, a particular data item might be locked by another transaction. In such a case the transaction requesting for the data item will have to wait till the transaction that have placed a lock on the data item releases it. A special graph *wait-for-graph* is constructed to store this information and the transaction requesting the data item is placed in a waiting state. The *wait-for-graph* is constructed to detect any deadlocks, which may arise due to inter-locking of transactions.

6.2.4 π -Database

The learned transaction execution patterns are maintained in this section of the TDB data structure. It provides means to make calls to *pattern recogniser and learner*. It is mainly used in the probabilistic conflict detection algorithms. The structure of the *transaction-rules/patterns* is discussed in detail in section 4.4.2.

6.2.5 Miscellaneous Data

There are miscellaneous data items that are mainly used by the internal transaction operations. These data structures include:

- Twin-transaction execution mode (FTM or MTM),
- TTM-ID,
- Connection/disconnection information, and
- Various other internal house keeping data structures.

6.2.6 Twin-Transaction Structure

A twin-transaction is internally represented by the twin-transaction structure (TTS). Different elements of a twin-transaction structure are shown in Figure 6.3 and are explained below.

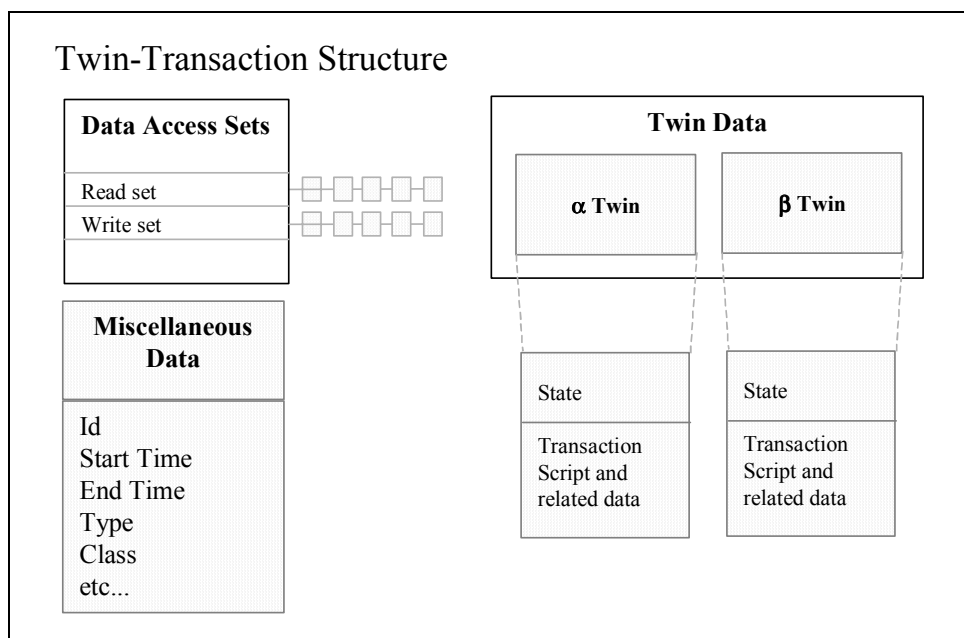


Figure 6.3 Main data structures in Twin-Transaction structure

6.2.6.1 Miscellaneous Data

TTS structure maintains a number of state parameters, which are required by different components of the twin-transaction system. The main ones are:

- TTM-ID (who submitted that transaction).
- Start and end time of transaction.
- Type of transaction (read, read/write, etc).
- Conflict resolution class (Abort, re-execute, application specific, etc).
- Application specific conflict resolver id.
- Conflict resolution method sequence.

6.2.6.2 Data Access Sets

Read and Write sets are the most important components of TTS. They are represented by a doubly linked list with each element containing a pointer to a DI-node. DI-node is the internal representation of a data item. DI-Node records the information about all the access operations performed by the twin-transactions on the data item.

6.2.6.3 Twin Data

When a twin-transaction structure is created it goes through the twinning process and two twins of the transaction are created, α and β twins. Each TTS represents both twins of a twin-transaction. The data elements that have different values for different twins are part of Twin-Data and consists of following elements:

- **State.**
Each twin of a transaction might be in different state.
- **Script.**
The transaction script of each twin is kept separate.

- Miscellaneous other internal information.

6.2.7 Data Item Node (DI-Node)

This data structure records the information about all the access operations performed by the twin-transactions. The locking and unlocking of data item is performed at this level.

Access to data items is provided using methods of DI-Node data structure. When a data item is locked, the transaction trying to access the data item is placed in the *wait-for-graph* and the data access method will not return. When the data item becomes available the data access method will return indicating that the data item is available now.

The *precedence graph* is also adjusted during data access operations. When a transaction updates a data item, a new version of the data item is created and the old version is stored in the *Access List*. This is to ensure a proper rollback when needed. The *Access List* can also be viewed as a storage place to store old versions of a data item.

The structure of DI-Node is shown in Figure 6.4.

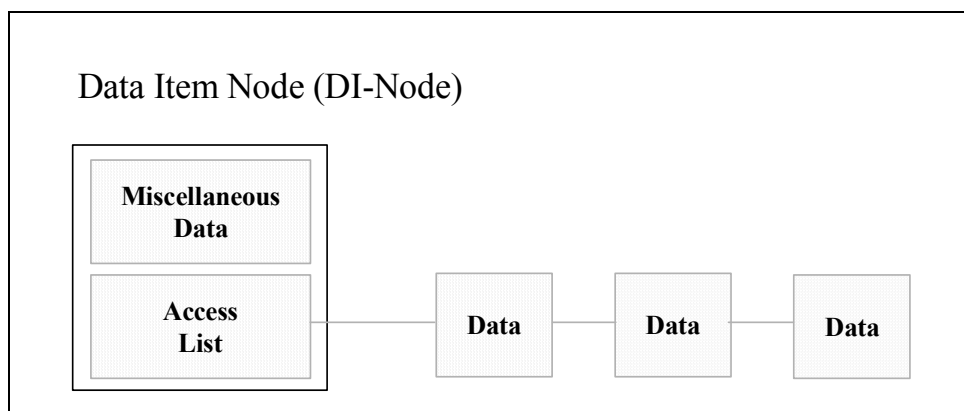


Figure 6.4 Item list Node (DI-Node) Structure

6.3 Maintaining Data Structures

Proper maintenance of data structures defined in section 6.2 is essential for keeping the twin-transaction system intact. They are also necessary for the resynchronisation process that is the heart of the twin-transaction system.

6.3.1 Transaction Lists

Transaction lists are maintained by TDB. Whenever a new transaction is created, it is added to a transaction list (or lists when transaction is twinned). TDB returns a transaction id to the caller. This id is used to perform operations within the same transaction. A transaction is moved from one transaction list to another depending on its state. TTS makes explicit calls to TDB to change transaction-state from one to another. This state change request then results in transaction being moved from one transaction list to another.

6.3.2 Read and Write Set Maintenance

The major bookkeeping activity performed by a twin-transaction system is done while executing a transaction. This involves recording *RAEDSET* and *WRITESET*. For every transaction operation, any or all of the following activities are carried out:

- DI-Node is modified,
- READSET and WRITESET are modified,
- precedence graph is modified, and
- Probability of success/failure is re-evaluated to take into account of the most recent operation performed by the transaction.

For recording READSET and WRITESET of a transaction, each read or write operation is performed on a DI-Node. The TTS structure is responsible for modifying its read and write set lists on successfully performing an operation on a DI-Node. The probability of success/failure is not re-evaluated after each operation, rather it is postponed till the point when the application makes an explicit request for this information.

The *precedence graph* maintenance is also performed at DI-Node level.

The linear search performed to search for a transaction in a transaction list or a data item in item list can become a performance bottleneck if the number of transactions in a transaction list or number of data items accessed by a transaction exceeds a certain threshold. More advanced data structures, such as hash table or binary trees, can be adopted in such situations.

6.4 Optimisation

The twin-transaction manager relies on many data structures that are explained throughout this chapter. The performance of the transaction system is dependent on better management of these data structures. A number of optimisations are incorporated to minimise the resources required. The major optimisations are discussed in the following sections.

6.4.1 DI-Node Space Management

The recording of READSET and WRITESET results in expanding DI-Nodes (data item nodes). The transaction system maintains these nodes to furnish the information about data access to the resynchronisation process. The possibility of a successful

resynchronisation process can be enhanced if the minor details of data access are recorded. On the other hand the space required to record these details would increase dramatically as the size of DI-Node increases.

To overcome limitations on DI-Node space management, a more precise control on what's required to be recorded is adopted. If an application-specific conflict resolver is specified for a transaction, then that resolver has the option to specify what should be recorded in a DI-Node.

Compression is a commonly used technique for reducing data size without information loss. In the twin-transaction system it can be highly beneficial to compress the DI-Node data lists since they might never be used if no conflicts are detected.

Another option is to inform the application committing a transaction about the size of DI-Node data and if the application decides, the DI-Node data is removed (will not be used during the resynchronisation process).

In current implementation, the data of a DI-Node is defined as a row of a table column (in relational database terms). The updates to a data item (DI-Node) can be performed on both whole of the data item (row) or on a column level. The DI-Node is expanded appropriately (based on whether the row is updated or the column).

6.4.2 $\pi(T)$ Management

As discussed in section 4.4.2.1, the size of $\pi(T)$ will increase for every new transaction.

If a twin-transaction manager does not go through the resynchronisation process for a

long time, the space required to store the ever-growing $\pi(T)$ can increase exponentially.

An optimisation is implemented that redefines $\pi(T)$ as:

$$\pi(T) = \pi(T_1) \cup \pi(T_2) \cup \dots \cup \pi(T_n)$$

Where $\pi(T)$ is defined as a union of $\pi(T_i)$ of transactions, on which a transaction T is dependent. The implementation keeps pointers to these sets and dynamically iterates through them when required.

6.5 Persistence and Crash Recovery

The transaction systems are designed to survive crashes (machine shutdown, power failure, run time errors etc.). It is imperative that a transaction system maintains its critical information on persistent storage and uses it to resume normal operation after a system restart. This section describes the data structures/mechanisms to record crucial information to a persistent storage media and recovering from crashes.

6.5.1 Persistent Data Structures

To make the data structures defined in previous sections persistent, a special persistent data manager is implemented. The persistent data manager makes system data persistent. Data creation/deletion operations are done via persistent data manager (PDM). Any data that is created via PDM is made persistent. The deletion of data is two-fold, deletion from memory and deletion from persistent store.

PDM ensures that persistent store is kept consistent. Each twin-transaction class (that is stored via PDM) derives from a special PDM_Object class and redefines two virtual

functions (READ and WRITE). PDM calls the WRITE function to instruct the class to convert itself into a stream of characters and READ function to instruct the class to build itself again from the character stream. Since many of the twin-transaction objects can have memory references to each other (pointers), PDM provides special means to convert these references from physical to logical and logical to physical references. This conversion is only performed during the load/unload process of persistent data. During a normal operation this translation is not required as all objects use physical memory addresses.

PDM takes periodic save-points, during which all modified data is transferred to persistent store. It also provides means that allow the twin-transaction manager to explicitly call for a save-point.

Almost all the important information included in the twin-transaction system such as TTS structure, including read set/write sets, DI-Nodes, and precedence graphs are stored via PDM. The intermediate data and the data that can be computed from other persistent data are not stored via PDM.

6.5.2 Crash Recovery

Since PDM loads information on-demand, during the start up process, the transaction system loads the necessary information and recreates internal data that is necessary to resume normal transaction services. If the transaction system was not stopped properly (a crash), the last consistent state is loaded before starting to recover transactions.

6.5.2.1 Recovering Transaction Lists

The recovery of the transaction lists is simple. The TTS structure is loaded and the state information about the two twins is retrieved and the twins are added to relevant transaction lists.

The transactions that are in *Running T_α List* require some extra work. This is due to the fact that the result of execution of transactions in this list is not known. In current implementation, a transaction in this list is aborted and operations that it performed on the data items (WRITESET) are reversed. This is possible as the modifications done by the transaction were recorded. Since all the transactions that are in *Running* state are aborted, the *wait-for-graph* is not stored on persistent store, as no transaction will be in *running* state on recovery and thus none will be waiting for any data items. All the locked data items are also released.

If the failure occurred during the resynchronisation process, then the recovery is slightly different on MTM than that of FTM. On an FTM, the FTM will process the β -twins that it has received and will wait for TTMs to start the resynchronisation processes again.

On an MTM, the resynchronisation process will stop. The MTM will go back to disconnected mode and reconnect with FTM at a later stage to attempt resynchronisation for rest of the transactions.

6.5.2.2 Recovering Precedence Graph

The precedence graph is the second major data structure of the twin-transaction system. It captures the interdependency of transactions. The precedence graph is

maintained/updated during and after the execution of a transaction. It is possible that in case of a failure, some transactions leave dangling edges in precedence graph. This is certainly true when one or more transactions are cleared/removed from transaction lists as described in previous section. In such a case, the dangling edges point to a non-existent transaction (transaction that has been cleared/removed).

In current implementation of the twin-transaction model, these dangling edges are also removed when a transaction is removed from a transaction list. On a TTM, the local concurrency control mechanism ensures that a *committed* (or *tentatively committed*) transaction cannot have dependency edge on a transaction in *running* state. This guarantees that by removing the transaction and its dangling edges from the precedence graph, the precedence graph becomes consistent.

6.5.2.3 Limitations

In the previous sections on recovery, the recovery mechanism assumes that all committed transactions have been transferred to a persistent store. It is possible, that a crash occurs during certain operations that leave the corresponding data in a bad state.

For example:

- Insertion of a node into precedence graph for a transaction. The precedence graph might be stored to persistent store before the transaction and crash occurs just after that. In such a case, the precedence graph contains a node that does not point to any transaction.
- A transaction has changed states (possibly from *running* to *tentatively committed* state). But before the change of state is recorded on persistent store, the crash occurs.

The work done between the last save point and point of crash will be lost. The recovery mechanism must clean up the precedence graph to remove nodes that are not linked to any transactions. But the recovery mechanism has no way of knowing that a transaction had changed states and the work done in such a case will be lost.

6.6 Resynchronisation Revisited

In sections 4.3, 4.4 and 3.3, the concurrency control and resynchronisation mechanisms are discussed. In this section some important implementation issues about concurrency control, resynchronisation and transaction validation that have not been addressed are discussed.

6.6.1 Data Item Maintenance

The very first step during the resynchronisation phase is to validate the data items on an MTM against their replicas on FTM. If any of the data items on FTM have changed, then new values should be loaded to make data items valid again. There are two approaches to establish the validity of data items:

- Compare the data items and if different update MTM with data item on FTM.
- Compare data item ids and if different update MTM with data item on FTM.

The first option, ie to compare whole data items is not practical due to high communication and processing costs. A better solution to implement first option is to skip the comparison and always load data items from FTM.

The second approach is more practical in the sense that only those data items are loaded from FTM who have changed. In current implementation, each data item is assigned an ITEM-ID that is composed of following fields:

- a) TTM-ID that created the data item.
- b) Timestamp of creation.

This ensures that each newly created data item gets a unique ITEM-ID. Each ITEM-ID has an UPDATE-ID that is composed of following fields:

- TTM-ID that updated the data item.
- Timestamp, that specify the modification time.

The attribute UPDATE-ID is used during the resynchronisation process to record the date/time of update and the id of TTM that updated that particular data item.

Next phase is to determine which data items (replicated on MTM) have been modified on FTM. The selection of an algorithm to perform this task is dependent on the following facts:

- Number of data items maintained by FTM. This number is usually huge as compared to number of data items replicated on MTM.
- Number of data items updated on FTM. Since MTM only need to receive those data items that are being updated.

The algorithm for synchronising data items that are not modified on MTM is given in Figure 6.5. The algorithm is divided into a number of steps that are carried out by MTM and FTM.

Steps on MTM

1. Select and count data items that are not modified on MTM.
2. Send count to FTM.
3. Receive response from FTM.
4. If FTM requires ID-Vector go to step 8.
5. Receive ID-vector from FTM.
6. Check UPDATE-ID of each item in ID-vector against UPDATE-ID of data items on MTM. If they match then item is removed from ID-vector.
7. Go to step 9.
8. Construct ID-vector of all selected data items (not updated on MTM). The ID-vector includes the UPDATE-ID.
9. Send ID-vector to FTM.
10. Receive new versions of data items.
11. End.

Steps on FTM

1. Count data items that are modified.
2. Receive count from MTM.
3. If FTM count < MTM Count go to step 8.
4. Ask for ID-Vector from MTM.
5. Receive ID-vector from MTM.
6. Check each UPDATE-ID of each item in ID-vector against UPDATE-ID of data items on FTM. If they match then item is removed from ID-vector.
7. Go to step 11.
8. Construct ID-vector of all updated data items. The ID-vector includes the UPDATE-ID.
9. Send ID-Vector to MTM.
10. Receive updated ID-Vector.
11. For each item that is in ID-vector, send the corresponding new version of data item to MTM.
12. End.

Figure 6.5 Algorithm to resynchronise unmodified data items.

On connection, MTM sends a count of data items that are not modified to FTM. FTM determines the number of items that have been modified and determines which count is smaller and based on that either sends the ID-Vector of all modified items or receives the ID-Vector of all items that are not modified. The next step of determining the items that are to be sent to MTM is carried out at either FTM or MTM end. UPDATE-ID of all those items that don't need to be updated is removed from the ID-Vector. All those items that have an entry in ID-Vector are sent to MTM. In a nutshell the algorithm

updates only those items on MTM whose UPDATE-ID has changed on FTM thus reducing the amount of network traffic between MTM and FTM.

6.6.2 Transaction Synchronisation

For all those data items that are modified on MTM, the MTM synchronisation mechanism tries to synchronise the transactions (that modified one or more data items) and MTM is updated with the result of the synchronisation process. The transaction synchronisation process operates in two modes, *connected* and *re-connected*. During the *connected* mode, the algorithm used to keep an MTM synchronised with FTM is given in Figure 6.6.

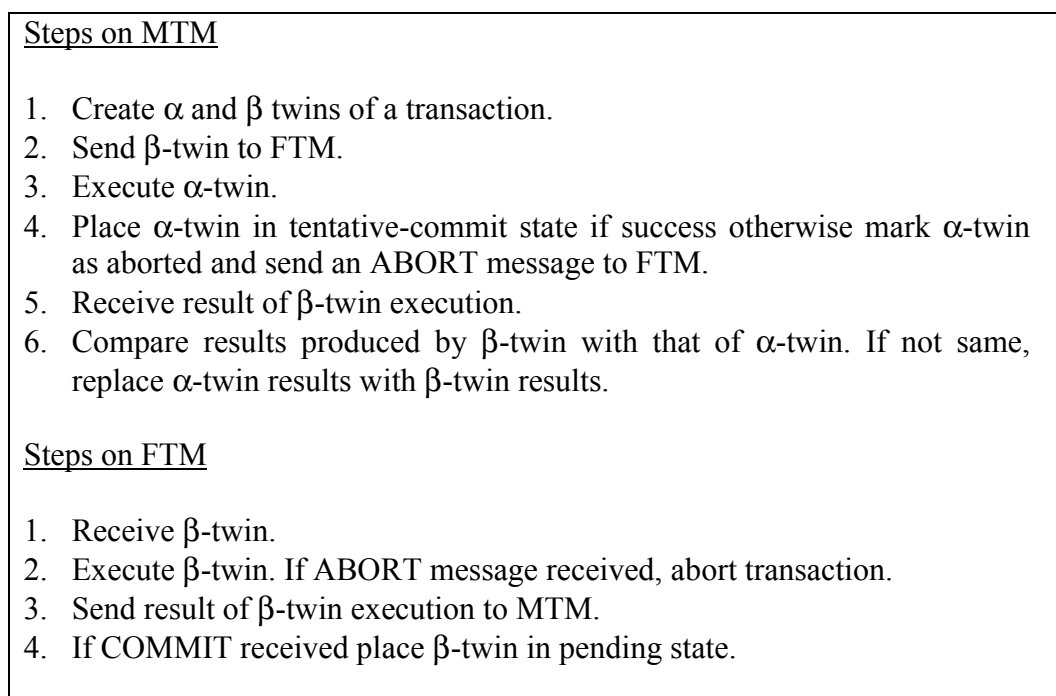


Figure 6.6 Algorithm to synchronise transactions in connected mode

As obvious from the algorithm, the α -twin stays in *tentative commit* state and β -twin stays in *pending* state. When β -twin transits to *resolved* state, only at that time α -twin can safely proceed to *commit* state. In *connected* mode, the transaction system sends the change-of-state information for β -twins straight away. Otherwise the change-of-state

information is send at the beginning of a resynchronisation process that is executed on re-connection.

During reconnection, the resynchronisation process must ensure that concurrent twin-transactions on different MTMs have not updated the same data item or have not accessed a data item that has been modified on another MTM. Any transactions that have done that must go through the conflict-resolution process.

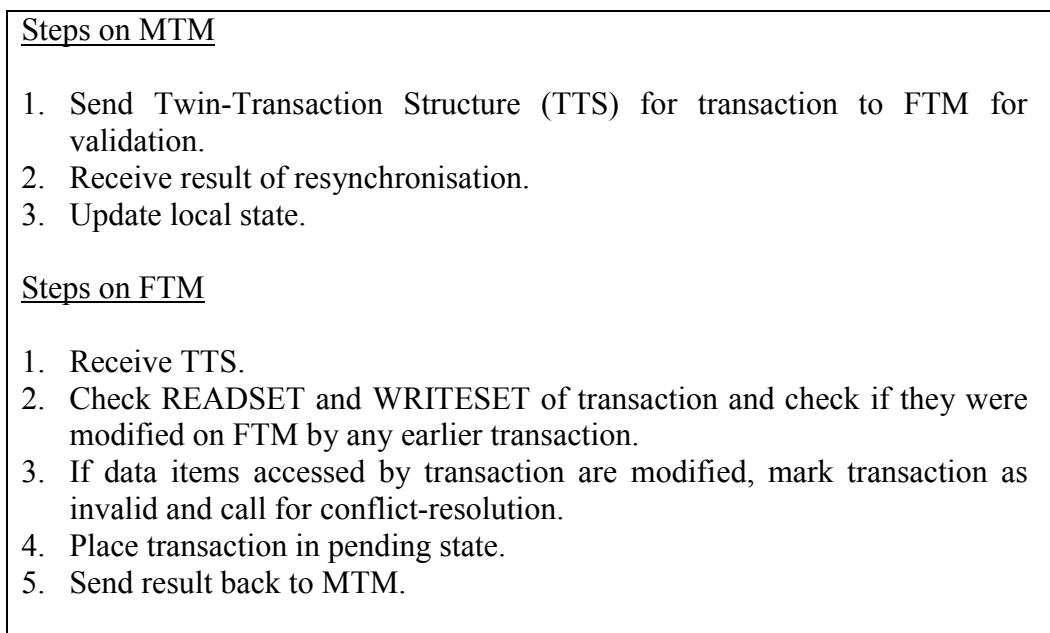


Figure 6.7 Algorithm to validate a transaction on reconnection

The algorithm to resynchronise an MTM on reconnection is given in section 5.1.2. The algorithm explains how transaction should be propagated and how they should be resolved in case they fail the validation process. The actual validation of a transaction is left to particular implementation of the twin-transaction system. In current implementation, a transaction is validated using the algorithm given in Figure 6.7. The algorithm resynchronise the MTM with FTM. This is done on transaction-by-transaction bases. The READSET and WRITESET of the transaction executed on an MTM are analysed to determine if the item accessed in a transaction was also involve in

a transaction on FTM. If such an item is encountered, the transaction resolution process kicks in and resolves the conflict (if any).

6.7 Transaction Script

A *transaction script* describes control flow and other execution strategies of a twin-transaction. The transaction script consists of *steps* that define the basic units of work in a transaction. In this implementation of the twin-transaction model, the transaction scripting language is a limited functional language. Each language construct is a function and returns a value. Some of the main elements are described below.

IF

Syntax:
 if (expression-1)
 (expression-2)
 else
 (expression-3)

Firstly *expression-1* is evaluated and if result is *true* (non-zero) *expression-2* is evaluated, otherwise *expression-3* is evaluated. The result of *if* function is either result of *expression-2* or *expression-3* (depending on *expression-1*).

WHILE

Syntax:
 while (expression-1)
 (expression-2)

The *expression-2* is executed until result of *expression-1* becomes 0.

READ

Syntax:
 read (item id)
 or

read(item id, col)

The item identified by *item id* is read into variable *var*. It returns the value just read.

WRITE

Syntax:

read (item id, var)

or

write(item id, var, col)

The item identified by *item id* is written with new value from variable *var*. It returns *true* if write is successful otherwise *false*.

ASSIGN

Syntax:

assign(var, expression)

A variable named *var* is assigned the value of *expression*. The result of the function is the value assigned to *var*.

Expressions

An expression in any of the above constructs can involve any of the language constructs including the following operators:

- Relational Operators
 - == equal
 - != not equal
 - > greater than
 - < less than
 - >= greater than or equal
 - <= less than or equal

- Logical Operators
 - & AND
 - | OR
 - ! NOT

6.8 Application Specific Resolvers

The current implementation of the twin-transaction model provides a few services to assist application specific resolvers. These services are listed in Table 6-1 and their usage/functionality is discussed after that.

Service	Parameters
hasAccessed	Transaction Id, Object Id, Access Type
doWrite	Object Id, new value
doRead	Object Id, FTM or MTM
preserveData	Object Id, FTM or MTM

Table 6-1 Services for application specific resolvers

hasAccessed

This service allows the resolver to determine whether a transaction has accessed a particular data item. The service returns *positive* value if the specified access has been performed otherwise it returns *zero*. A *negative* value indicates that an error occurred during the execution of the service. The returned value corresponds to a specific error code.

doWrite

The application resolver can instruct the twin-transaction manager to update a particular

data item with the specified new value. This is usually the case when an application resolver has succeeded in resolving a conflict.

doRead

During the resolving phase, the application specific resolver will need to read the FTM or MTM copy of a data item to determine the nature of conflict. This service performs this task.

preserveData

Rather than performing a doWrite service call, the application specific resolver can indicate, using this service, which data item to be preserved (FTM or MTM) and the other one is discarded.

6.9 Summary

The chapter presented different aspects of the implementation of the twin-transaction model. The data structures used and the optimisations performed are discussed. The persistence management and crash recovery mechanisms are also explained. The transaction script elements are also presented.

The next chapter discusses the performance of twin-transaction model and presents the results of test runs. These results are analysed to test the applicability of the twin-transaction model.

Chapter 7

Evaluation

A working implementation of the twin-transaction model has been developed to measure its use and performance. Simulated applications to test the viability of the twin-transaction model are tested under the developed twin-transaction system. In this chapter the simulated applications and the data from the implementation is provided to quantify the cost of the twin-transaction model.

7.1 Overview/System Status

The twin-transaction model is a transaction system and requires a number of components to be programmed before proper evaluation. The first version of the twin-transaction implementation did not have any transaction execution functionality and was merely simulating the conflicts. This version became operational in June 1998. It served as a basic prototype for experimenting and fine-tuning various design and implementation alternatives.

In late 1998, a major system overhaul and re-implementation were conducted. This implemented system supports most of the twin-transaction model except:

- Pattern recogniser and learner are not fully dynamic.

- Global certification consistency guarantee is not provided.
- Transaction script lacks functionality for resolving conflicts in β -twin transaction.

The main objectives of the implementation are:

- Realisation of a data management system that supports data item storage and transaction processing on those data items.
- Multiple instances of the system running on different (or same) machines to simulate multiple twin-transaction systems with connection/disconnection.
- Support for capturing transaction history and activation of pattern recogniser and learner.
- Resynchronisation of the twin-transaction systems on re-connection.

The current implementation is complex and took longer to code due to the following main factors:

1. The need to operate the client in two different modes for conflict representation and resolution.
2. The need to operate in both client and server modes.
3. To provide G1SR consistency guarantee, a complete transaction history is maintained.
4. Failures during consistency validation also added a great deal of complexity to the overall system.

Each data item, under the current implementation, is treated as a separate entity. A unique identifier (called ITEM-ID) is assigned to each data item. This identifier is used by transactions to operate on a specific data item. It is also used during resynchronisation of replicas of the same data item.

7.2 Evaluation Approach

The purpose of the dissertation is to verify the hypothesis that the twin-transaction model substantially improves consistency support for transaction driven mobile applications during disconnected mode of operation. Previous chapters have shown how twin-transaction model executes transactions and maintains consistency among clients and servers. This chapter focuses on evaluating the performance and resource cost for supporting twin-transactions.

The overall evaluation approach is to execute controlled experiments. The transaction performance and resource costs are evaluated. The current implementation of the twin-transaction model is also used to build a test application to determine the applicability of the twin-transaction model for a practical application (section 7.7).

The evaluation of the twin-transaction model is done on the following lines:

- Transaction performance
- Resource cost management
- Resynchronisation performance (Conflict detection and resolution)

The transaction system is run on two machines with different configurations as shown in Table 7-1.

Name	CPU	Memory	Disk	Operating System
FTM	200 MMX	64 MB	2.1 GB	Windows NT
MTM	200 MMX	32 MB	1.2 GB	Windows 95

Table 7-1 TTM Configurations

7.3 Transaction Performance

The design and implementation of the twin-transaction model dictates that as the number of activities performed by the transaction manager increases, the performance overhead also increases. The key performance evaluation question is:

“Can twin-transaction system offer satisfactory overall performance for executing common applications? Is the performance overhead caused by twin-transaction operations acceptable?”

Applications executing transactions will have to pay a performance cost. This performance cost is part of the total cost of running a twin-transaction. To determine the performance overhead caused by added operations of the twin-transaction model, the performance of the twin-transaction system is recorded by running transactions with and without the twin-transaction operations and the results are compared.

7.3.1 Methodology

The performance overhead comes from a variety of resources. These are discussed below:

- Maintenance of transaction READSET and WRITESSET require operations on linked lists. These operations include:

Searching

Inserting, and

Deleting

- Maintaining transaction history that includes recording of transaction operations and its environment.
- Transaction pattern learning and probabilistic detection of possible conflicts.
- Maintenance of other internal structures, such as precedence graphs.

To quantify these overheads, transactions are designed to trigger each kind of overhead. The transactions are executed on TTMs in disconnected mode. The elapsed time is measured and compared with and without twin-transaction operations. This is done for each transaction.

7.3.1.1 Transactions

A variety of transactions are used to evaluate the performance overhead of the twin-transaction model. Each of them involves executing a certain workload on the twin-transaction manager. These transactions are described in Table 7-2.

The values of n and m are varied. For the transactions with parameters n and m , the data items to be accessed are selected pseudo-randomly. Once transaction is built (after the selection of data items), the transaction is executed a number of times (not consecutively but interlaced among other transactions). This is normally the case in a transaction environment, where a typical kind of transaction gets executed a number of times. This produces the data for *pattern recogniser and learner* and is later on used in *probabilistic conflict detection* calculations.

Transaction Name	Description
READ_ALL	Reads all data items that are available on a twin-transaction system.
READ(n)	Reads n data items.
WRITE(n)	Write n data items.
READWRITE(n,m)	Reads n data items and writes m data items.
INSERT(n)	Insert n new data items.
DELETE(n)	Delete n data items.

Table 7-2 Benchmark Transactions

In a transaction system, transactions interact with each other by operating on the same data items (in some serial schedule). Since the twin-transaction system captures the transaction history and constructs the precedence graph, the level of *interaction* of transactions is controlled by a constant (ϵ) expressed as a percentage of total number of transactions in the transaction benchmark. A higher value dictates more interactions among transactions.

7.3.2 Results

The transaction benchmark is constructed in the first phase of experiment. A list of transactions is constructed using the transactions of Table 7-2 and varying the constants n and m . The constant ϵ is also defined to control the level of interaction of transactions.

In the second phase, the transactions in the transaction benchmark are executed and the elapsed time for each kind of transaction is measured. These measurements are done for normal operations of the transaction system (without twin-transaction operations) and with twin-transaction (TT) operations. The overhead caused by twin-transaction operations is calculated as follows:

$$Overhead = \frac{\left(\frac{\sum_{i=1}^n T_{twin}^i}{n} \right) - \left(\frac{\sum_{i=1}^n T_{normal}^i}{n} \right)}{\left(\frac{\sum_{i=1}^n T_{twin}^i}{n} \right)} \times 100$$

Equation 7-1 Twin-transaction overhead

Where T_{twin} is the elapsed time to execute transaction T with twin-transaction operations and T_{normal} is the elapsed time to execute transaction T in the normal operation of the system.

7.3.2.1 Experiments

In each experiment, the average elapsed time for each kind of transaction is listed for normal operation of the transaction system and with twin-transaction operations (TT). The overhead of twin-transaction operations is calculated using Equation 7-1.

The transaction $ReadWrite(n,n)$ reads n data items and write n data items, where transaction $ReadWrite(n,m)$ reads n data items and writes m data items such that $m < n$.

The average time to execute each kind of transaction is measured in seconds and is shown in the result table of each experiment.

7.3.2.1.1 Experiment One

The results of this experiment are shown in Table 7-3. The interaction of transactions is kept at 10% ($\epsilon = 10\%$), ie, only 10% of transactions accessed some common data item. A total of 700 transactions are executed. The distribution of each kind of transaction is

also given as a percentage in Table 7-3. The results are an average of 20 runs of the experiment.

Transaction	(%)	MTM			FTM		
		Normal	With TT	Over-head	Normal	With TT	Over-head
Read All	4%	160.5	180.1	10.8%	159.1	177.9	10.5%
ReadWrite(n,n)	16%	3.0	3.4	11.7%	2.9	3.25	10.9%
ReadWrite(n,m)	16%	4.1	4.7	10.8%	2.7	3.0	10.0%
Insert(1)	16%	1.5	1.6	6.2%	1.5	1.52	1.3%
Delete(1)	16%	0.5	0.54	7.4%	0.5	0.51	1.9%
Insert(10)	16%	9.2	9.7	5.1%	8.1	8.3	2.4%
Delete(10)	16%	3.0	3.2	6.2%	3.0	3.1	3.2%

Table 7-3 Performance overhead ($\epsilon = 10\%$)

The results on both MTM and FTM indicate that the performance overhead of *insert* and *delete* operations is less than those of *read* and *write* operations. The overhead for *insert* is low because the probability of a conflict resulting from an insert is low. The low overhead of the *delete* operation can be attributed to the fact that once an item is deleted, no other operations can be performed on that item unless a conflicting transaction is executed on another TTM. The maximum performance overhead incurred by twin-transaction operations is around 11%, which is for read/write transactions.

7.3.2.1.2 Experiment Two

To further examine the performance overhead, the percentage of Insert and Delete transactions are decreased and separate Read and Write transactions are added to the transaction benchmark. The results are shown in Table 7-4. The results are an average of 20 runs of the experiment.

In this experiment the overhead caused by *insert* and *delete* has increased as compared to Experiment One. The increase in *delete* overhead is due to the increase in the number

of *read/write* transactions that are being executed and are causing conflicts. The increase in *insert* overhead is also due to the increase in the number of *read/write* transactions as size of history log is increasing. In this experiment too, the performance overhead incurred due to twin-transaction operations is around 11%.

Transaction	(%)	MTM			FTM		
		Normal	With TT	Over-head	Normal	With TT	Over-head
Read_All	4%	165.0	185.2	10.9%	161.0	180.9	11.0%
ReadWrite(n,n)	16%	3.1	3.5	11.4%	3.0	3.4	11.7%
ReadWrite(n,m)	16%	4.0	4.5	11.1%	3.9	4.4	11.3%
Read(1)	14%	1.5	1.67	10.1%	1.41	1.57	10.2%
Write(1)	14%	1.5	1.69	11.2%	1.42	1.58	10.5%
Insert(1)	9%	1.6	1.7	6.2%	1.4	1.51	7.2%
Delete(1)	9%	0.7	0.72	2.8%	0.7	0.75	6.7%
Insert(10)	9%	9.5	9.9	4.0%	9.4	10	6%
Delete(10)	9%	3.4	3.7	8.1%	4.1	4.5	8.9%

Table 7-4 Performance overhead ($\epsilon = 10\%$)

7.3.2.1.3 Experiment Three

In experiments two and three, the level of interaction of the transactions was kept at 10% ($\epsilon = 10\%$). In experiment three, the transactions for experiment two are reconstructed with $\epsilon = 30\%$. The results are shown in Table 7-5. The results are an average of 20 runs of the experiment.

The transaction execution time has not changed much and the overall performance overhead incurred is still around 11%.

7.3.2.2 Discussion

The above sets of experiments cover a broad range of workloads in disconnected operation. The performance overhead for *read* and *write* transactions of experiments is

presented in graphical form in Figure 7.1. The observed performance degradation for twin-transaction operations is between 10 ~ 12%.

Tranascion	(%)	MTM			FTM		
		Normal	With TT	Over-head	Normal	With TT	Over-head
Read All	4%	166.0	187.0	11.2%	165.1	184.1	10.3%
ReadWrite(n,n)	16%	3.11	3.53	11.9%	3.0	3.41	12.0%
ReadWrite(n,m)	16%	4.2	4.7	10.6%	3.9	4.43	11.9%
Read(1)	14%	1.4	1.57	10.8%	1.34	1.51	11.3%
Write(1)	14%	1.5	1.7	11.7%	1.34	1.50	10.7%
Insert(1)	9%	1.6	1.71	6.4%	1.45	1.55	6.5%
Delete(1)	9%	0.6	0.72	7.4%	0.7	0.76	7.9%
Insert(10)	9%	10.1	10.7	5.6%	9.6	10.1	5%
Delete(10)	9%	4.2	4.6	8.7%	4.2	4.7	10.6%

Table 7-5 Performance overhead ($\epsilon = 30\%$)

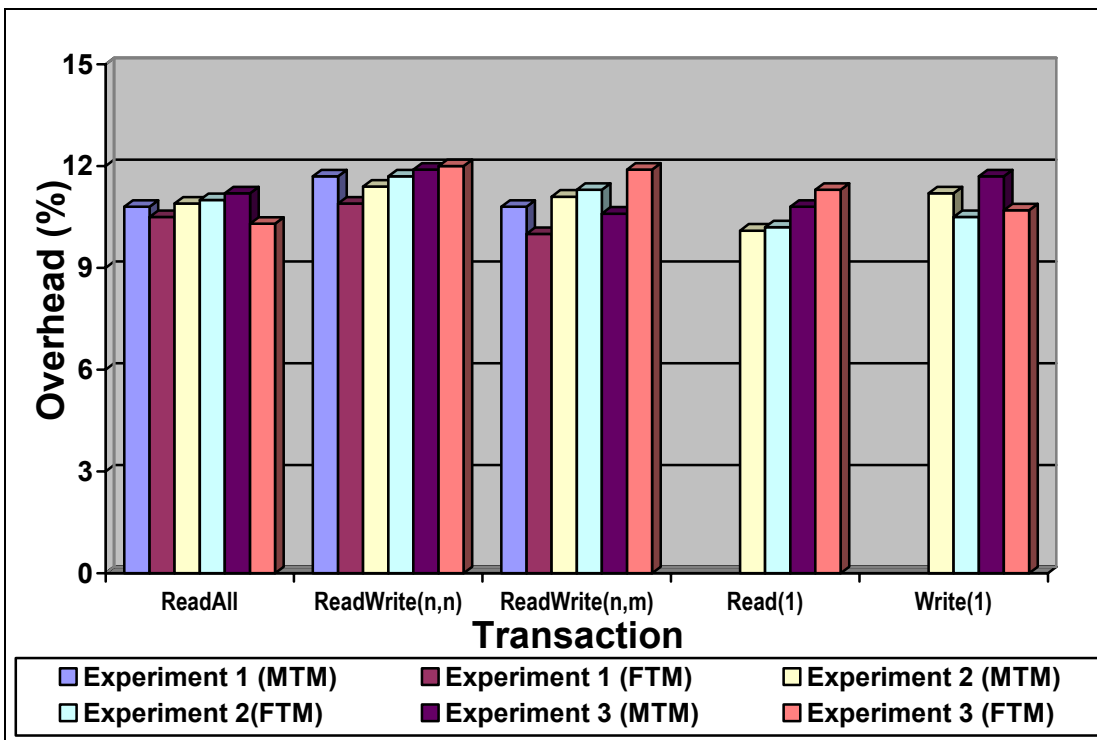


Figure 7.1 Performance overhead

The performance decrease is due to *pending* transactions. This is due to the fact that when a transaction operates on a data item that is produced (mutated) by a transaction in *pending* state, additional internal work needs to be done. This is obvious with

performance overhead difference between experiments two and three when the interaction of transaction was increased, resulting an increase in the number of transactions operating on data items that are produced by pending transactions.

Although the performances of both MTM and FTM have decreased, the advantage is that MTM can now work in disconnected mode.

The performance can be worse when a transaction tries to access an object that is currently locked by an ongoing transaction; it will have to block until the two-phase locking protocol completes. Since this performance degradation will occur in any transaction system and is not caused by twin-transaction operations, in the above experiments, this situation is not captured (transactions are not executed concurrently). The evaluation of this performance overhead is also excluded due to the reason that a disconnected MTM is typically operated by a single user and the likelihood of executing concurrent transactions performing conflicting accesses on shared data is very low.

A set of separate experiments to measure the performance overhead of transaction execution in a connected environment is carried out due to two main reasons:

1. The twin-transaction system is intended for use mainly on disconnected mobile workstations.
2. By design, the performance overhead of connected transaction execution is basically the same as that of disconnected execution. Even when MTM is connected, all operations of a transaction are logged and committed in the same fashion as if MTM is disconnected.

Some of the performance overhead is due to certain implementation specific choices and could be improved by using better alternatives. In summary, there is sufficient evidence to believe that the twin-transaction system can be realised at modest performance cost.

7.4 Resource Cost Measurements

The previous section focussed entirely on the CPU overhead of using the twin-transaction system. In reality, the twin-transaction system also incurs other overheads, such as:

- Memory
- Disk space
- Network bandwidth

In this section, the resource costs associated with twin-transaction operations are examined. The key questions is:

“Which system resources are subject to increased consumption by the twin-transaction system?”

The network bandwidth is consumed during the resynchronisation process. The memory and disk space usage on an FTM also increases during the resynchronisation process. Thus on an FTM, the resource cost measurement during the resynchronisation process should also be carried out. The costs associated with the resynchronisation process are discussed in section 7.5. In this section the focus is on disk and memory resource overheads for normal operation of the twin-transaction system.

7.4.1 Methodology

The disk space usage of the twin-transaction system increases due to following housekeeping operations:

- Transactions are logged.
- Data item state change is logged.
- Transaction pattern is captured and stored.
- Precedence graph is maintained.

The persistence data manager (PDM) is responsible for storing the transaction history and data item mutations. Storage of transaction precedence graph to persistent store (disk) is also the responsibility of PDM. The pattern recogniser and learner (PRL) maintains the transaction pattern database (π -Database) on disk and in memory.

Current implementation of the twin-transaction system stores all the data items through persistence data manager (except for π -Database). The disk resource overhead caused by the twin-transaction system is:

$$\begin{aligned} \text{Disk space overhead} = & \\ & \text{Space used by } \pi\text{-Database} + \\ & \text{Space used by PDM} - \\ & \text{Space used by PDM to store data items.} \end{aligned}$$

Equation 7-2 Disk space overhead

The space required to store the data items is subtracted from the total disk space used, as at least this much space will be used by any transaction system to store data items.

The memory usage of the twin-transaction system also increases due to *precedence graph*, which is kept in memory. Any pending transaction that has mutated a data item must also stay in memory until it becomes *obsolete* (section 4.2.1). Thus the total memory usage overhead is calculated as:

$$\begin{aligned} \text{Memory overhead} = & \\ & \text{Memory used by precedence graph} + \\ & \text{Memory used by pending transactions} \end{aligned}$$

Equation 7-3 Memory overhead

Some memory is used by PRL to maintain some necessary data structures in memory but it only consumes a small amount of memory and is not included in the calculation of memory overhead.

7.4.2 Results

The experiments two and three, described in section 7.3.2.1, are used to simulate a test run of the twin-transaction system. The memory and disk space usage is recorded. The disk space overhead is shown in Figure 7.2 and memory overhead is shown in Figure 7.3.

For a disconnected operation of three hours, 12MB-disk space overhead is incurred. The space overhead increases consistently as MTM remains in disconnected mode. This disk space overhead is due to the logging of extra information required by the twin-transaction model.

The memory overhead for disconnected operation of three hours is around 2MB. The memory overhead at the start of the disconnected operation increases sharply but after some time the increase in memory starts to flatten. This behaviour is due to following reasons:

- At the start of execution, the transactions and *precedence* graph are constructed causing rapid increase in memory overhead.
- After the twin-transaction system is running for some time, the transactions start to become *obsolete* and can be removed from memory.
- The steady increase in memory overhead is forced by *precedence* graph, which is kept in memory in full.

The resource cost is dependent on the size of the precedence graph. Better management of *precedence* graph can minimise the memory usage by transferring parts of *precedence* graph to disk when certain nodes are no longer needed for execution of future twin-transactions (nodes belonging to *obsolete* transactions).

	Memory overhead (100 KB)								
	20	40	60	80	100	120	140	160	180
Trace 1	2	4	9	11	14	16	19	21	22
Average	3.2	6.4	11	14.6	16.6	18.4	19.6	21.6	22.8
Trace 2	3	7	11	15	17	18	18	22	23
Trace 3	4	8	13	18	19	20	19	23	23
Trace 4	2	5	10	13	15	17	20	21	22
Trace 5	5	8	12	16	18	21	22	21	24

Table 7-6 Memory overhead

	Disk space overhead (100 KB)								
	20	40	60	80	100	120	140	160	180
Trace 1	10	17	25	39	51	67	80	103	120
Trace 2	11	19	26	38	54	69	81	99	119
Trace 3	6	15	23	37	59	71	83	98	115
Trace 4	7	16	24	32	55	70	81	97	109
Trace 5	9	21	29	40	54	73	84	101	115

Table 7-7 Disk space overhead

Its evident from the conducted set of experiments that the system resource cost associated with disconnected transaction usage is modest. Considering the rapidly growing disk and memory capacity, this increase in the disk and memory usage for the twin-transaction system should be acceptable.

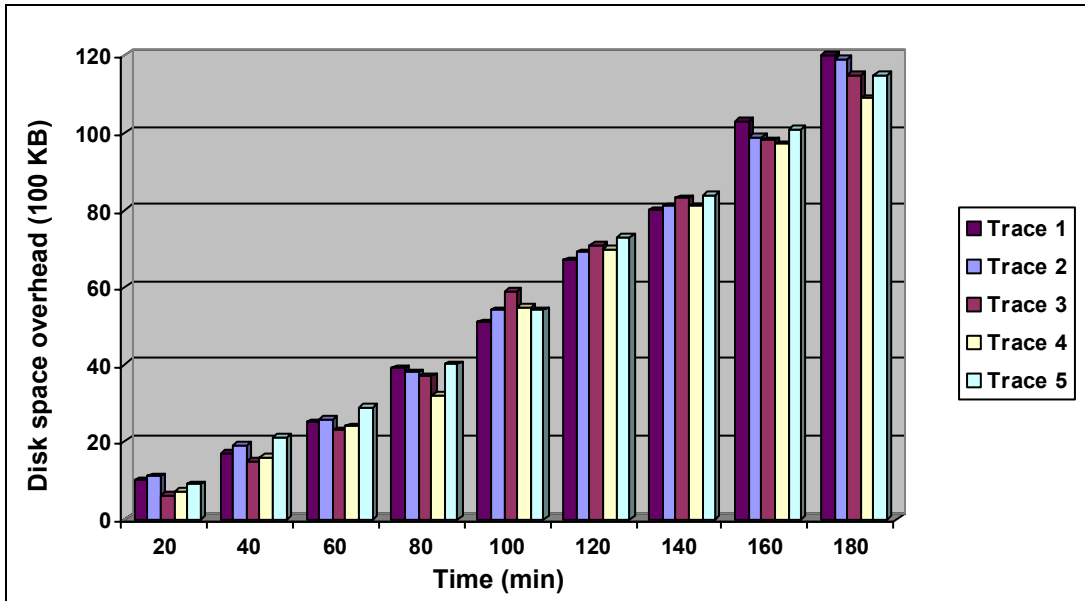


Figure 7.2 Disk space overhead

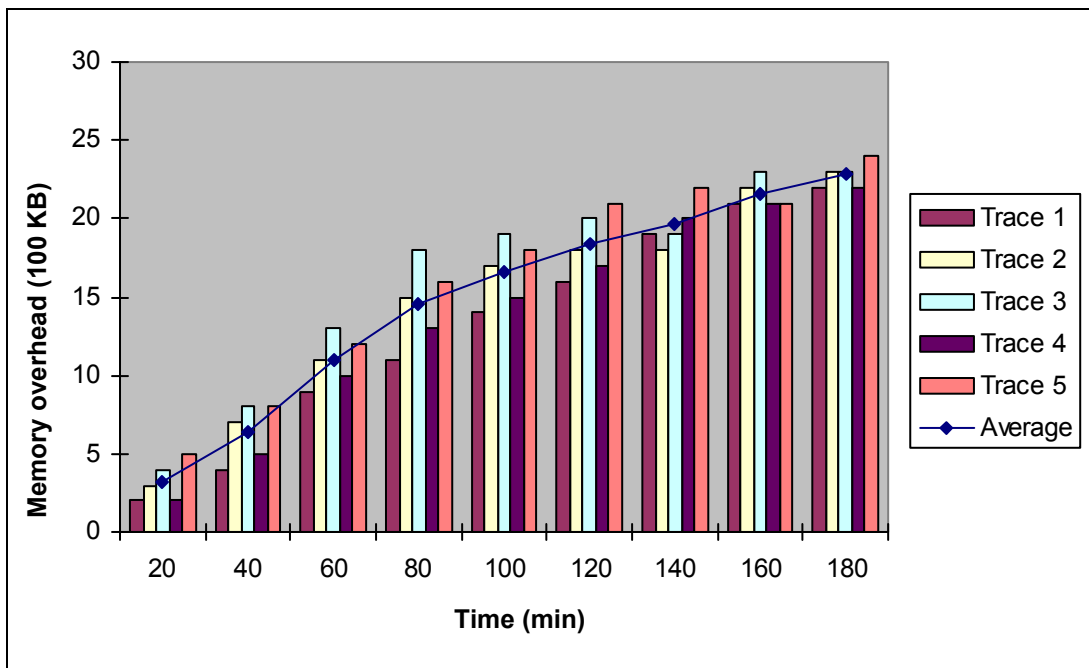


Figure 7.3 Memory overhead

7.5 Resynchronisation Performance

In section 7.4, the resource cost on a TTM is measured. In this section, the resource cost on FTM due to resynchronisation and *pending* transactions is measured. The performance cost associated with resynchronisation and *pending* transactions is also analysed.

There are three kinds of resources associated with the resynchronisation operation:

1. Network traffic
2. Disk and memory usage
3. System performance (CPU usage)

When an MTM connects with an FTM and resynchronises itself, the network traffic increases due to transfer of transaction history from MTM to FTM and transfer of updated data items to MTM.

The resynchronisation process is executed on FTM. This in turn increases CPU usage and a decrease in system performance. The disk and memory usage also increases when MTM transactions are transferred to FTM, where they are kept until they are resolved and results are transferred back to MTM.

Since the resynchronisation process does not effect an MTM operation and its resource usage, FTM load and network traffic due to resynchronisation process is measured. This data is used to analyse the resynchronisation performance and resource cost. The FTM load refers to the total amount of FTM CPU, disk and memory usage.

7.5.1 Methodology

There are a number of activities carried out by FTM that consume resources:

- Transaction validation
- Transaction resolution if required
- Transaction integration into FTM state
- Data item version comparison and exchange of updated data items.

In the first phase of experiments, the measurements of system resource cost incurred by the resynchronisation process are discussed. In the second phase the network cost associated with transfer of transaction history and exchange of updated data items is measured and discussed.

7.5.2 FTM Resource Cost Measurements

The MTM state after execution of experiment 3 (7.3.2.1) is used to perform the resynchronisation process. The Table 7-8 shows the total elapsed time, disk and memory usage of MTM to perform the resynchronisation process.

Run Number	Elapsed Time (sec)	Disk Usage (100 KB)	Memory Usage (KB)
1	223	20	21
2	252	33	30
3	231	41	33
4	240	43	37
5	233	50	36
6	230	42	33
7	237	44	32
8	250	35	29
9	243	39	40
10	233	45	37

Table 7-8 FTM Resource Cost

Table 7-8 shows an almost consistent *elapsed time* for the resynchronisation of MTM for each run of experiment three. The disk usage increases steadily, whereas the memory usage during the resynchronisation process also increases steadily. This result was expected as the transactions, even though are synchronised, are kept in *pending* state (on disk) and the *precedence* graph (in memory) expands to cater for the inter-dependency of transactions that are now in *pending* states.

In this experiment only one MTM client was used, therefore although transactions were validated, no conflict was detected (as no other MTM resynchronised itself with FTM). That indicates that Table 7-8 does not represent costs associated with transaction resolution. To determine the cost of transaction resolution and the number of transactions that cannot be accepted, the experiment was conducted again but this time two MTM clients were used, executing conflicting transactions. The results are shown in Table 7-9.

Run Number	Elapsed Time (sec)	Disk Usage (100 KB)	Memory Usage (KB)	Transactions Rejected
1	233	30	24	0%
2	301	34	33	7%
3	320	40	45	9%
4	353	44	51	11%
5	360	49	50	12%
6	341	51	49	8%
7	352	53	47	9%
8	369	60	39	11%
9	372	67	57	10%
10	340	55	43	11%

Table 7-9 FTM Transaction Resolution Costs

The results shown in Table 7-9 indicates that resource costs have increased along with the time required for resynchronising. This is a direct result of transaction resolution activity. The number of transactions that cannot be resolved are around 10~11%,

although the transaction system tries to resolve transactions using re-execution and executing application specific resolvers.

The percentage of transactions that cannot be resolved is dependent on the nature of the applications. When MTMs execute a number of conflicting transactions, it is not always possible for a resolver to achieve a consistent state satisfying all the conflicting transactions (by executing transaction again, or by using application specific resolvers). It is expected that the percentage of transactions that can not be resolved due to execution of conflicting transactions on different MTMs is around 15%.

7.5.3 Network Traffic

On reconnection, MTM resynchronises itself with FTM. This involves exchange of information between MTM and FTM. MTM sends the transaction history log to FTM so that its transaction history should be integrated into transaction history of FTM. FTM sends back a response indicating the result of such integration (resynchronisation). The network traffic between MTM and FTM increases due to this exchange of information. In this section, the network traffic overhead caused by twin-transaction system is presented. The overhead is calculated using the following equation:

$$\text{Network Overhead} = \text{Total network traffic} - \text{Data item exchange traffic}$$

Equation 7-4 Network overhead

The *data item exchange traffic* is caused by exchange of data items between MTM and FTM. It is assumed that any data management system that tries to keep two network partitions in a synchronised state will at-least have to consume this much resource. Thus

the overhead caused by the twin-transaction system is calculated using the above equation and is shown in Table 7-10.

Table 7-10 shows that the overhead in network traffic is around 50%. The overhead is high due to following reasons:

- The network traffic caused by data item version comparison is not subtracted from the *total network traffic*.
- Apart from *data exchange*, database management systems do exchange state information to keep data in synchronised form. Network traffic caused by these operations is not taken into account.

Run Number	Total Network Traffic (KB)	Data Item Exchange Traffic (KB)	Overhead
1	1003	701	43.1%
2	1050	699	50.2%
3	1105	800	38.1%
4	1233	730	68.9%
5	1145	636	80.0%
6	1099	711	54.5%
7	1043	703	48.4%
8	1155	763	51.4%
9	1140	779	46.3%
10	1063	685	55.2%

Table 7-10 Network traffic overhead

Due to above mentioned reasons, in reality the network traffic overhead caused by the twin-transaction system is less than that shown in Table 7-10. The overhead can further be decreased with a better representation of the transaction history.

7.6 Probabilistic Conflict Detection/Estimation

Both MTM and FTM also compute the probability of conflict for transactions. The performance overhead caused by this activity is taken into account in the previous

sections/experiments. In this section the cost of deferring the execution of a transaction due to probabilistic conflict detection (PCD) and resynchronising it is compared with the cost to execute the transaction and resynchronise it. The two costs for experiment 3 (section 7.3.2.1) are given in Table 7-11, Table 7-12, Table 7-13 and Table 7-14. The cost calculations are done by calculating the following costs:

- A. Cost of deferring transaction execution and then resynchronisation.**
- B. Cost of executing transaction and then resynchronisation.**

The total resources saved is equal to $B - A$. A negative value indicates that more resources are used.

Since resources of an MTM are considered precious as compared to those on FTM, the cost of using a resource on MTM is much higher as compared to usage of the same on FTM.

Run Number	MTM			FTM		
	A	B	B-A	A	B	B-A
1	11	12	1	21	21	0
2	7	7	0	30	30	0
3	8	10	2	33	31	2
4	10	9	-1	37	37	0
5	6	7	1	36	35	1

Table 7-11 Probabistic conflict detection: memory performance

Run Number	MTM			FTM		
	A	B	B-A	A	B	B-A
1	30	29	-1	20	20	0
2	21	21	0	33	33	0
3	25	31	6	41	45	4
4	20	35	15	43	42	-1
5	24	33	9	50	55	5

Table 7-12 Probabistic conflict detection: disk performance

Not every transaction execution can be deferred. The transaction originator has the control over whether to defer the execution of a transaction or not.

Run Number			
	A	B	B-A
1	43.1	55.1	10
2	50.2	50.2	0
3	38.1	47.2	9.1
4	68.9	74.1	6.2
5	80.0	86.2	6.2

Table 7-13 Probabistic conflict detection: network performance

Run Number	Execution on MTM			Resynchronisation		
	A	B	B-A	A	B	B-A
1	193	203	10	233	249	16
2	178	178	0	252	252	0
3	165	170	5	231	233	2
4	159	177	18	240	267	27
5	183	205	22	233	256	23

Table 7-14 Probabistic conflict detection: performance

In run 2, no transaction was deferred which resulted in zero gain or loss in resources. In all other runs, there is a definitive decrease in the consumption of resources on both ends (MTM and FTM) but more importantly less resources are consumed on MTM. This is directly related to the fact, that when a transaction causes conflict, it will result in more work on FTM (apart from the work already done on MTM). If it can be predicted and taken into account, the resource consumption on both ends will decrease.

7.7 Contact Manager - A Test Application

A simple *Contact Manager* application is defined here. It represents a class of applications that are more suited for the twin-transaction system. The test application requirements are defined as follows:

An organisation has a number of salesmen who are responsible for meeting customers and selling goods. The meetings with customers are booked by a number of operators in the main office. Each salesman carries a mobile workstation and down loads his/her meeting plans from the main office server.

A salesman can also book a future meeting with the customer by recording an appointment on his mobile workstation, which they up-load to main office server.

The appointments made by salesmen and operators might collide causing a conflict.

The structure of the application is driven by the following facts:

- There is only one kind of transaction *making an appointment*.
- The *Contact Manager* application is like a calendar for each salesman.
- MTM should maintain a calendar for each salesman whereas FTM should maintain calendar for every salesman (for operators).
- The operators access FTM directly and run their transactions on it and salesmen make appointments on MTMs.
- The calendars on MTM and FTM are resynchronised periodically.
- At the most, the conflict can only arise between one particular MTM and FTM. This is due to the fact that each MTM maintains a separate calendar. Thus after resynchronisation, each MTM becomes fully

synchronised with FTM and none of the transactions for a particular salesman calendar are left in *pending* state.

This application was selected due to its simplicity, yet applicability to the twin-transaction model. The features include:

- The conflict can only occur between FTM and MTM.
- The calendar for each salesman can be treated as one data item that is replicated.
- Application specific conflict detection can be utilised to pinpoint the conflicting access (date and time).
- Application specific conflict resolution can be used to resolve a conflict.

A total of 10 calendars (for 10 salesmen) are maintained. This results in 10 MTM sessions and 1 FTM session. Up to 20% transactions are run on FTM and the rest on MTM. That is, assuming that salesmen make most of the appointments. The resynchronisation interval (Δt) is equal to 24 hours.

The resource costs (including costs incurred due to the resynchronisation process) of FTM and an MTM are shown in graphical form in Figure 7.4. The network overhead of synchronisation over a number of runs and the number of transactions found in conflict and being resolved is shown in Table 7-10.

As shown in Figure 7.4, the disk and memory usage on MTM and FTM increases steadily during normal operation. On each resynchronisation operation, both disk and memory usage increases sharply and then drops back to normal. This fluctuation in disk

and memory usage for MTM is consistent. After each resynchronisation process, the MTMs resource usage drops to a minimal level, but for FTM, the graph shows that disk and memory usage does not drop to a minimal level. Rather, the usage keeps on increasing steadily. This behaviour is attributed to following facts:

- Every new transaction execution results in slight increase in the resource usage. These transactions are being executed during the time the resynchronisation with an MTM is taking place.
- Although transactions related to one calendar are resolved, transactions executed on other calendars (FTM is maintaining not one but all the calendars) are still in *pending* state.
- The transaction log is maintained till the next savepoint.

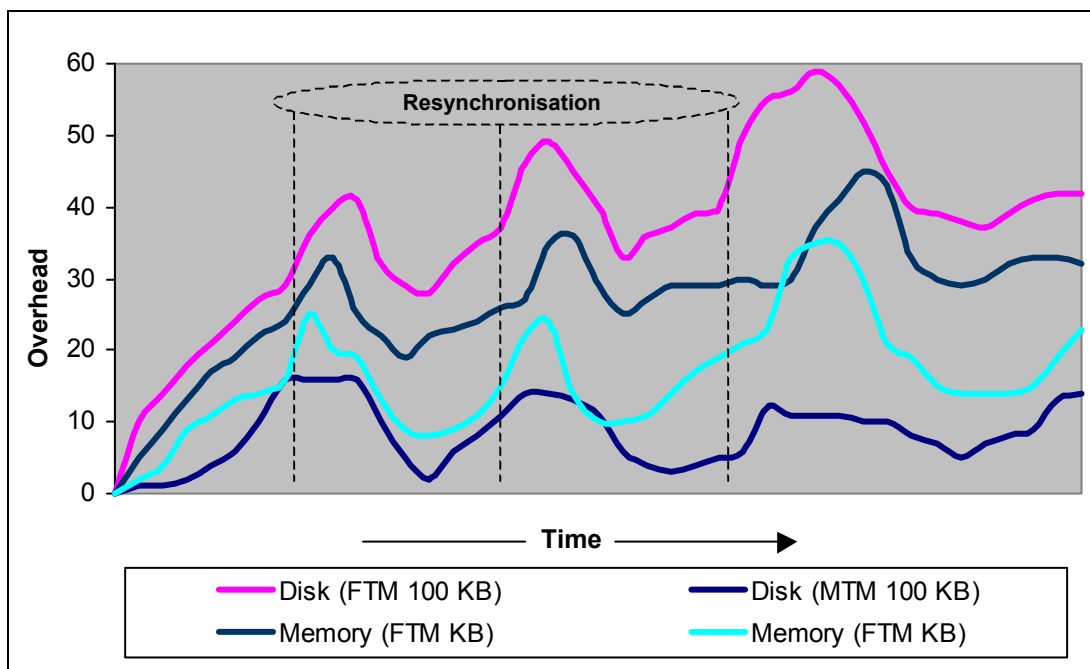


Figure 7.4 Disk and memory resource costs

Table 7-15 shows the network traffic overhead caused by each resynchronisation process. An application specific resolver (ASR) is used to resolve in conflict transactions (same time slot allocated to two customers). The resolver simply allocates

another time slot to the customer. In run 9, the number of transactions in conflict was 8, but the number of transactions reported to be in conflict by ASR is 10. This happened due to following reason:

In some instances, since the application resolver is working on a transaction-by-transaction basis, the new time slot allocated to a customer might be in use by another transaction (on MTM) yet to be validated. When that transaction is validated, the result produced by ASR becomes in conflict with this transaction. Thus reporting that more transactions are in conflict.

Resynchronisation Run Number	Network Overhead (KB)	Transactions Resolved
1	50	2
2	52	3
3	55	6
4	60	1
5	57	5
6	33	6
7	29	7
8	40	6
9	46	10
10	42	3

Table 7-15 Network overhead and transactions resolved

7.8 Further Evaluation

So far, twin-transaction system performance and resource cost (including network resources) are measured using controlled experiments. The behaviour of the twin-transaction system under a real life application is only tested under a very simple application (section 7.7). The behaviour of the twin-transaction system under a more sophisticated application is still yet to be seen. Moreover, some of the results could be further validated or adjusted with more data/experiments.

Chapter 8

Conclusion

This dissertation has described the twin-transaction model to facilitate disconnected operations of a mobile client and maintain data consistency during disconnected and connected modes of operations of a mobile host. The central idea consists of four basic elements:

- Logging of transaction histories.
- Optimistic enforcement of serialisability based consistency requirement.
- Automatic detection and resolution of conflicts.
- Probabilistic estimation of conflicts during disconnected mode of operation.

Flexible conflict resolution mechanisms and integration of application-specific semantics not only help in resolution of conflicts but also in the detection of conflicts. The concept of twinning of a transaction is used to incorporate application-specific conflict resolution semantics that are used during resynchronisation of TTMs.

The design, implementation, experimentation and evaluation of the twin-transaction model indicates the following conclusions:

- The twin-transaction model is a feasible way of addressing the data inconsistency problems caused by disconnected operation of mobile hosts.
- Initial evidence from controlled experiments and test application shows that twin-transaction model's conflict detection and resolution mechanisms are effective.
- Due to lack of usage experience, a definitive conclusion on the useability of the twin-transaction model needs to be deferred until more applications using the twin-transaction model are developed and their performance is evaluated.
- The probabilistic conflict estimation helps the transaction manager and users to get ready for things to come during the resynchronisation process.
- The twin-transaction models practicality is demonstrated by its ability to support the developed test application, which incorporates the basic elements of a transaction-oriented application.

The twin-transaction model is more complex than was expected. This can be attributed to the inherent difficulties of maintaining and propagating transactions in a mobile environment. The design trade-offs for a clean model of client/server state synchronisation and transaction resolution significantly increased complexity and its associated system development and maintenance costs. Such design knowledge could not have been obtained without the complete process of development of an actual twin-transaction system.

8.1 Contributions

This thesis is one of the first attempts to develop a transaction system that addresses the data inconsistency problems caused by partitioned read/write conflicts and provide G1SR guarantees. The twin-transaction model is designed solely for the purpose of improving consistency in mobile data access. The specific contributions of this thesis research can be classified into the following areas:

1. *Conceptual Analysis*

- Data inconsistencies caused by partitioned read/write conflicts within the context of a mobile computing environment.
- Analysis of serialisability-based consistency model to detect data inconsistencies.
- Applicability analysis of replication methods for mobile computing environment.

2. *System Design*

- The design of a new abstraction for both transactions and data items that captures the abstraction of mobile computing environment: Replication of data and replication of transactions to manage inconsistencies arising due to partitioned access of data.
- Integration of application-specific knowledge at the very root of conflicts, namely the transaction. This is done by an explicit definition of dissimilar transaction twins.

- Provision for external application specific resolver and conflict detectors to help automate the resynchronisation process.
- A conflict representation scheme that provides resolvers convenient access to information relevant to resolving conflicting transactions.
- Probabilistic conflict estimation to help applications determine the possibility of success on resynchronisation.
- The design of a *transaction pattern* database and abstractions to compare *similar* transactions during probabilistic conflict estimations.
- Usage of incremental transaction propagation model to simplify the tasks of resolver programming and manual conflict resolution.

3. *Implementation*

- A successful combination of optimistic concurrency control across MTM and FTM with strict local two phase locking for transaction processing.
- A transaction script interpreter in the form of a functional programming language that enables convenient transaction specification for both twins of a transaction.
- A special interface for application-specific resolvers that support access to transaction patterns and replicas of data item on both MTM and FTM.
- A resolver mechanism that supports automatic transaction re-execution or invocation of an application specific resolver.
- A persistent data manager to store data on persistent store and recover from system crashes.

4. *Experimentation and Evaluation*

- Empirical measurements based on controlled experiments to determine the performance and resource costs of the twin-transaction model.
- The development of a test application to determine the performance and resource costs of the twin-transaction model under an actual application.
- The demonstration of effectiveness of probabilistic conflict estimation in reducing client space cost for those transactions that are doomed to be in conflict during resynchronisation.

8.2 Future Work

A number of implementation enhancements have been suggested in previous chapters. These enhancements were not coded due to time constraints. Also a few minor features logically belonging to the twin-transaction system have not been programmed. For example a full dynamic probabilistic conflict estimation learner and pattern recogniser is not yet implemented. This section describes additional implementation extensions that will make the current twin-transaction system more complete. In addition, a few other areas worth investigating are discussed.

Data Item Structuring

Current implementation of the twin-transaction model was carried to achieve two important goals:

1. The system should be able to run controlled experiments.
2. The system should provide adequate services for the development of mobile applications.

Although current implementation achieved these two goals, it lacks the functionality of a true database management system. It only provides one flat view of data items. It is desirable for most applications that DBMS provide some form of structuring of data items eg similar to relational databases (tables, rows and columns).

Consistency Criterion

Current implementation does not provide GC (global certificates) consistency validation criterion. The consistency validation mechanisms need to be adapted in order to provide GC consistency validation criterion (in place of G1SR).

Integration of heuristic systems

To decrease the amount of work required during the resynchronisation process due to transactions that are in conflict, the twin-transaction mechanism tries to estimate the probability of conflict for each transaction. This estimation is based on transaction patterns/rules that are stored in the pattern database. The current implementation makes analytical decisions using the pattern database. Although some optimisations are implemented in pattern matching, the use of heuristic systems (eg. expert systems) should be investigated to minimise the time and resources required to determine the probability of conflict.

Application-Specific Resolver

Although the twin-transaction system heavily relies on β -twin of a transaction to resolve conflicts (execution of β -twin), the use of application-specific resolvers can also play a key role in the twin-transaction system. However, there is a lack of support for the programming of application-specific resolvers in the current implementation. Although

experimental resolvers are developed for the testing of the twin-transaction system, the approach has been ad-hoc. Generally speaking, the research on how to best support resolver development is still in its infancy. One possible direction in supporting application-specific resolvers can be taken to increase the functionality that is provided by the interpreter developed to handle transaction scripts. Adding a few enhancements to transaction script can achieve this.

Scalability

Twin-transaction model fits well in a small-scale mobile system. However, the model does not scale well since there is a single FTM. To improve the scalability of the twin-transaction system, multiple FTMs are required, each of them sharing the load of MTM transactions and carrying out resynchronisation processes.

When there are more than one FTM available, an MTM might choose to resynchronise with any of them. Further to complex the matters, it might choose different FTM each time it needs to resynchronise. To keep data and transactions consistent and conflict free, two options are available:

- I. A resynchronisation process between different FTMs must also be carried out from time to time.
- II. A mechanism is required to get a consensus for each transaction from the set of FTMs (accept/reject).

The option II requires that FTMs (depending upon the mechanism to reach a consensus for each transaction) should communicate with each other on a transaction-by-

transaction basis when an MTM is going through the resynchronisation process on any of the FTMs. This should be kept in mind that multiple MTMs on different FTMs might be performing this operation. If sufficient bandwidth is available for communication between different FTMs then this option will produce better results and a shorter lifecycle for transactions.

The option I is feasible for situations where FTMs cannot stay connected with each other all the time or the bandwidth between them is not available to support option II. But detailed study and experiments are required to confirm these speculations.

8.3 Final Remarks

This dissertation has shown that it is practical and feasible to use the twin-transaction mechanism to support transaction-based mobile computing environment for some classes of applications, while keeping the consistency of shared data that is accessed by multiple mobile workstations (in connected and disconnected modes). Three basic ideas made it possible: keeping a log of transactions, optimistic enforcement of serialisability-based consistency requirement and automatic detection and resolution of conflicts along with the use of probabilistic estimation of conflicts during disconnected mode of operation. In addition the use of dissimilar twins and application-specific resolvers proves to be invaluable in consistency maintenance of mobile data access.

Bibliography

- [AAC92] M. Ahamad, M. H. Ammar and S. Y. Cheung. Replicated Data Management in Distributed Systems. In *Readings in Distributed Computing Systems*, 1992.
- [AB87] M. P. Atkinson and O. P. Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, 19(2), June 1987.
- [ABG88] R. Alanso, D. Barbara and H. Garcia-Molina. Quasi-copies: Efficient Data Sharing for Information Retrieval Systems. In *Proceedings of the Second International Conference on Extending Data Base Technology*, Venice, Italy, March 1988.
- [AB93a] A. Acharya and B. R. Badrinath. Delivering Multicast Messages in Networks with Mobile Hosts. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, Pennsylvania, May 1993.
- [AB93b] A. Acharya and B. R. Badrinath. *A Framework for Delivering Multicast Messages in Networks with Mobile Hosts*. Rutgers DCS TR-310, 1993.

- [AH87] T. Andrews and C. Harris. Combining Language and Database Advances in an Object-Oriented Development Environment. In *OOPSLA '87 Conference Proceedings, ACM*, October 1987, published as SIGPLAN Notices, 23(11), November 1988.
- [AK89] S. Abiteboul and P. C. Kanellakis. Object Identity as a Query Language Primitive. *ACM SIGMOD Record*, 18(2), June 1989.
- [AK93] R. Alonso and H. Korth. Database System Issues in Nomadic Computing. In *Proceedings of ACM SIGMOD Conference*, May 1993.
- [AP91] B. Awerbuch and D. Peleg. Concurrent Online Tracking of Mobile Users. In *Proceedings of ACM SIGCOMM Symposium on Communication, Architecture and Protocols*, October 1991.
- [Ast79] M. M. Astrahan, et al. An Overview of System R: A Relational Database System. *IEEE Computer*, 13(4), 1979.
- [AT89] A. El-Abadi and S. Toueng. Maintaining Availability in Partitioned Replicated Databases. *ACM TODS*, 14(2), June 1989.
- [Atk94] M. Atkinson. Persistent Foundations for Scalable Multi-Paradigm Systems. In M.T. Özsu, U. Dayal and P. Valdureiz, editors. *Distributed Object Management*. Morgan Kaufmann Publishers, 1994.
- [BAI93] B. R. Badrinath, A. Acharya and T. Imielinski. Impact of Mobility on Distributed Computations. *ACM Operating Systems Review*, 27(2), April 1993.

- [BAI94a] B. R. Badrinath, A. Acharya and T. Imielinski. Checkpointing Distributed Applications on Mobile Computers. In *Proceeding of 3rd IEEE International Conference on Parallel and Distributed Information Systems*, October 1994.
- [BAI94b] B. R. Badrinath, A. Acharya and T. Imielinski. Structuring Distributed Algorithms for Mobile Hosts. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, June 1994.
- [BB95] A. Barkre and B. R. Badrinath. I-TCP: Indirect TCP for mobile hosts. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, May 1995.
- [BBG89] C Beeri, P. A. Bernstein and N. Goodman. A Model for Concurrency in Nested Transaction Systems. *Journal of ACM*, 36(2), April 1989.
- [BBIM93] B. R. Badrinath, A. Bakre, T. Imielinski and R. Marantz. Handling Mobile Clients: A Case for Indirect Interaction. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, October 1993.
- [BG81] P. A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2), 1981.
- [BG84] P. Bernstein and N. Goodman. An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases. *ACM TODS*, 9(4), December 1984.

- [BG94] D. Barbara and H. Garcia-Molina. Replicated Data Management in Mobile Environments: Anything New Under The Sun? *IFIP Transactions*, vol. A-44, 1994.
- [Bha86] B. K. Bhargava, editor. *Concurrency Control and Reliability in Distributed Systems*. Van Nostrand Reinhold, 1986.
- [BHG87] P. A. Bernstein, V. Hadzilacos and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley. Reading, Massachusetts, 1987.
- [BHM90] P. A. Bernstein, M. Hsu and B. Mann. Implementing Recoverable Requests Using Queues. *ACM SIGMOD*, 1990.
- [BHMC90] A. P. Buchmann, M. Hornick, E. Markatos and C. Chronaki. Specification of a Transaction System for Distributed Active Object System. In *Proceedings of the Workshop on Transactions for Object Oriented System held in conjunction with OOPSLA '90*, Ottawa, Canada, October 1990.
- [BHN87] P. A. Bernstein, V. Hadzilacos and N. Goodman. *Concurrency Control And Recovery In Database Systems*, Addison-Wesley, Reading, Massachusetts, 1987.
- [BI92a] B. R. Badrinath and T. Imielinski. Querying in Highly Mobile Distributed Environments. In *Proceedings of the 18th International Conference on Very Large Databases*, August 1992.

- [BI92b] B. R. Badrinath and T. Imielinski. Replication and Mobility. In *Proceedings of the Second Workshop on the Management of Replicated Data*, November 1992.
- [BI92c] B. R. Badrinath and T. Imielinski. Locating Strategies for Personal Communications Networks. In *IEEE Globecom 92, Workshop on Networking of Personal Communications Applications*, 1992.
- [BI93] B. R. Badrinath and T. Imielinski. Data Management for Mobile Computing. In *SIGMOD Record*, 22(1), March 1993.
- [BI94] B. R. Badrinath and T. Imielinski. Mobile Wireless Computing: Solutions and Challenges in Data Management. *Communications of the ACM*, October 1994.
- [BI94] B. R. Badrinath and T. Imielinski. Sleepers and Workaholics: Caching Strategies in Mobile Environment. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, 1994.
- [BKK85] F. Bancilhon, W. Kim and H. Korth. A Model of CAD Transactions. In *Proceedings of the 11th International Conference on Very Large Databases*, 1985.
- [BK91] N. S. Barghouti and G. E. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 23(3), September 1991.
- [BMS92] Y. Breitbart, H. Garcia-Molina and A. Silberschatz. Overview of Multidatabase Transaction Management. Technical Report CS-TR-92-1432, Department of Computer Science, Stanford University, May 1992.

- [BÖ90] K. Barker and M.T. Özsu. Concurrent Transaction Execution in Multidatabase Systems. In *Proceedings of COMPSAC'90*, 1990.
- [BP85] A. P. Buchmann and C. Perez de Celis. An Architecture and Data Model for CAD Databases. In *Proceedings of the 11th International Conference on Very Large Databases*, 1985.
- [BR87] B. R. Badrinath and K. Ramamritham. Semantics Based Concurrency Control: Beyond Commutativity. In *Proceedings of 3rd International Conference on Data Engineering*, 1987.
- [BR88] B. R. Badrinath and K. Ramamritham. Synchronizing Transactions on Objects. *IEEE Transactions on Computers*, 37(5), May 1988.
- [Bre89] R. Bretl, et al. The GemStone Data Management System. In W. Kim and F. H. Lochovsky, editors. *Object-Oriented Concepts, Databases and Applications*, ACM Press / Addison-Wesley, 1989.
- [BSR80] P. A. Bernstein, D. W. Shipman and J. B. Rothnie. Concurrency Control in a System for Distributed Databases (SDD-1). *ACM Transactions on Database Systems*, 5(1), March 1980.
- [Buc90] A. Buchmann. Modeling Heterogeneous Systems as a Space of Active Objects. In *Proceedings of the 4th International Workshop on Persistent Object Systems*, 1990.

- [Buc92] A. Buchmann, M. T. Özsu, M. Hornick, D. Georgakopoulos and F. M. Manola. A Transaction Model for Active Distributed Object Systems. In Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*, Morgan Kaufmann Publishers, Inc., 1992.
- [But87] M. H. Butler. Storage Reclamation in Object-Oriented Database Systems. In *Proceedings of the 1987 ACM-SIGMOD International Conference on Management of Data*, ACM, New York, 1987.
- [Cac93] R. Caceres, F. Douglass, K. Li and B. Marsh. Operating System Implications of Solid-State Mobile Computers. Technical Report MITL-TR-56-93, 1993.
- [Cer91] S. Ceri, A. W. H. Maurice, M. K. Arthur and P. Samarati. A Classification of Update Methods for Replicated Databases. Technical Report, Computer Science Department, Stanford University, 1991.
- [CH91] S. Ceri, M. Houtsma, A. Keller and P. Samarati. A classification of update methods for replicated databases. Technical Report STAN-CS-91-1392, Stanford University, 1991.
- [Chr93] P. K. Chrysanthis. Transaction Processing in Mobile Computing Environment. In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, October 1993.
- [CI94] R. Caceres and L. Iftode. The Effects of Mobility on Reliable Transport Protocols. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, June 1994.

- [CK94] H. K. Chiou and W. Korfhage. Efficient Global event Detection. In *Proceedings of The 14th International Conference on Distributed Computing Systems*, June 1994.
- [Con89] R. C. H. Connor, et. al. An Object Addressing Mechanism for Statically Typed Languages with Multiple Inheritance. In *OOPSLA '89 Conference Proceedings*, ACM, October 1989, published as SIGPLAN Notices, 24(10), October 1989.
- [CP84] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems*. McGraw-Hill Book Company, 1984.
- [CR90] P. K. Chrysanthis and K. Ramamritham. ACTA: A Framework for Specifying and Reasoning About Transaction Structure and Behavior. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1990.
- [Dav82] S. Davidson. An Optimistic Protocol for Partitioned Distributed Database Systems. Phd Thesis, Princeton University, October 1982.
- [Dav84a] S. Davidson. Optimism and consistency in partitioned distributed database systems. *ACM Transactions on Database Systems*, 9(3), September 1984.
- [Dav84b] S. Davidson. Optimism and Consistency in Partitioned Distributed Database Systems. *ACM Transactions on Database Systems*, 9(3), 1984.
- [Day93] U. Dayal. An Activity/Transaction Model for a Distributed Multi-Service System. In M.T. Özsu, U. Dayal and P. Valdureiz, editors. *Distributed Object Management*. Morgan Kaufmann Publishers, 1994.

- [DBM88] U. Dayal, A. Buchmann and D. McCarthy. Rules Are Objects Too: A Knowledge Model For An Active Object-Oriented Database System. In *Proceedings of the 2nd International Workshop on Object Oriented Database Systems*, 1990.
- [DE89] W. Du and A. Elmagarmid. Quasi-Serializability: A Correctness Criterion for Global Concurrency Control in InterBase. In *Proceedings of the 15th International Conference on Very Large Data Bases*, 1989.
- [Dem94] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer and B. Welch. The Bayou Architecture: Support for Data Sharing among Mobile Users. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, December 1994.
- [Dem87] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the ACM Sixth Symposium on Principles of Distributed Computing, SIGACT-SIGOPS*, August 1987.
- [DGG95] K. R. Dittrich, S. Gatzui and A. Geppert. The Active Database Management System Manifesto: A Rulebase of ADBMS Feature. In *Proceedings of the 2nd Workshop on Rules in Database Systems (RIDS)*, Athens, Greece, Springer LNCS, September 1995.
- [DGP90] A. Downing, I. Greenburg and J. Peha. OSCAR: An Architecture for Weak Consistency Replication. In *Proceedings of the IEEE PARABASE-90*, March 1990.

- [DGS85] S. B. Davidson, H. Garcia-Molina and D. Skeen. Consistency in Partitioned Networks. *ACM Computing Surveys*, 17(3), September 1985.
- [DHB97] M. H. Dunham, A. Helal and S. Balakrishnan. A Mobile Transaction Model that Captures Both the Data and Movement Behaviour. *ACM Journal on Special Topics in Mobile Networks and Applications (MONET)*, 1997.
- [DKB93] F. Douglass, P. Krishnan and B. Marsh, Systems Issues in Mobile Computing. Technical Report MITL-TR-61-93, July 1993.
- [DTK88] A. Dan, D. Towsley and W. Kohler. Modeling the Effects of Data and Resource Contention on the Performance of Optimistic Concurrency Control Protocols. In *Proceedings of the 4th International Conference on Data Engineering*, 1988.
- [Du89] W. Du, A. K. Elmagarmid, Y. Leu and S. Osterman. Effects of Local Autonomy on Global Concurrency Control in Heterogeneous Distributed Database Systems. In *Proceedings of the 2nd International Conference on Data and Knowledge Systems for Manufacturing and Engineering*, October 1989.
- [Duc92] D. Duchamp. Issues in Wireless Mobile Computing. In *Proceedings of the Third Workshop on Workstation Operating Systems*, 1992.
- [Edw97] W. K. Edwards, E. D. Mynatt, K. Petersen, M. J. Spreitzer, D. B. Terry and M. M. Theimer. Designing and Implementing Asynchronous Collaborative Applications with Bayou, 1997.

- [Elm90] A. Elmagramid, Y. Leu, W. Litwin and M. Rusinkiewicz. A Multidatabase Transaction Model for InterBase. In *Proceedings of the 16th International Conference on Very Large Data bases*, 1990.
- [Elm92] Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*, Morgan Kaufmann Publishers, Inc., 1992.
- [ES83] D. L. Eager and K. C. Sevcik. Achieving Robustness in Distributed Database Systems, *ACM Trans. on Database Systems*, Vol 8, No. 3, Sep. 1983.
- [Esw76] K. P. Eswaran, J. Gray, R. Lorie and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of ACM*, 19(11), 1976.
- [EVT88] F. Eliassen, J. Veijalainen and H. Tirri. Aspects of Transaction Modelling for Interoperable Information Systems. In *Interim Report of the COST 11ter Project*, 1988.
- [Fai95] M. Faiz. Database Replication in Mobile Computing Environment. Masters Thesis, Department of Computer Technology, Monash University, Australia, July 1995.
- [Fis87] D. H. Fishman, et al. Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, 5(1), January 1987.

- [FÖ89] A. A. Farrag and M. T. Özsu. Using Semantic Knowledge of Transactions to Increase Concurrency. *ACM Transactions on Database Systems*, 14(4), December 1989.
- [FZ94] G. H. Forman and J. Zahorjan. The challenges of Mobile Computing, *IEEE Computer*, 17(4), April 1994.
- [FZ95] M. Faiz and A. Zaslavsky. Database Replica Management Strategies in Multidatabase Systems with Mobile Hosts. In *Proceedings of 6th International Hong Kong Computer Society Database Workshop: Database Reengineering and Interoperability*, Hong Kong, March 1995.
- [Gan93] S. Ganguly and R. Alonso. Query Optimization for Energy Efficiency in Mobile Environments. In *Proceedings of the 1993 International Workshop on Foundations of Models and languages for Data and Objects*, Aigen, Austria, 1993.
- [Gif79] D. K. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the ACM Seventh Symposium on Operating System Principles*, December 1979.
- [GK88] H. Garcia-Molina and B. Kogan. Node Autonomy in Distributed Systems. In *Proceedings of IEE International Symposium on Databases and Distributed Systems*, December 1988.
- [GL91] R. Golding and D. Long. Accessing Replicated Data in Large Scale Distributed Systems. *International Journal in Computer Simulation*, 1(2), June 1991.

- [Goo83] N. Goodman, et al. A Recovery Algorithm for a Distributed Database System. In *Proceedings of the Second SCM SIGACT-SIGMOD Symposium on Database Systems*, Atlanta, GA, March 1983.
- [GP86] V. Gilor and R. Popescu-Zeletin. Transaction Management in Distributed Hetrogeneuos Database Systems. *Information Systems*, 11(4), 1986.
- [GR93] J. Gray and A. Reuter. *Transaction Processing (Concepts and Techniques)*. Morgan Kaufman publishers. 1993.
- [Gra81] J. N. Gray. The transaction concept: Virtues and limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases*, September 1981.
- [Gup89] A. Gupta, editor. *Integration of Information Systems: Bridging Hetrogeneous Databases*. IEEE Press, 1989.
- [Had88] V. Hadzilacos. A Theory of Reliability in Database Systems. *Journal of ACM*, 35(1), January 1988.
- [HB91] A. R. Hurson and M. W. Bright. Multidatabase Systems: An Advanced Concept in Handling Distributed Data. *Advances in Computers*, Academic Press. Inc., 1991.
- [Her86] M. Herlihy. A Quorum-Consensus Replication Method for Abstract Data Types. *ACM Transactions on Computer Systems*, 4(1), February 1986.
- [Her90] M. Herlihy. Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types. *ACM Transactions on Database Systems*, 15(1), March 1990.

- [HLM88] M. Hsu, R. Ladin and D. McCarthy. An Execution Model for Active Database Management Systems. In *Proceedings of the 3rd International Conference on Data and Knowledge Bases*, 1988.
- [Hol88] B. Holtkamp. Preserving Autonomy in a Hetrogeneous Multidatabase System. In *Proceedings of the 12th International Computer Software and Application Conference*, 1988.
- [HW88] M. Herlihy and W. Weihl. Hybrid Concurrency Control for Abstract Data Types. In *Proceedings of the 7th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1988.
- [IB93] T. Imielinski and B. R. Badrinath. Data Management for Mobile Computing. *SIGMOD Record*, 22(1), March 1993.
- [IB94] T. Imielinski and B. R. Badrinath. Wireless Mobile Computing: Challenges in Data Management. *Communications of the ACM*, October 1994.
- [IDM91] J. Ioannidis, D. Duchamp and Jr. G. Q. Maguire. IP-Based Protocols for Mobile Internetworking. Technical report CUCS-006-91 Department of Computer Science, Columbia University, New York, 1991.
- [IK95] T. Imielinski and H. F. Korth. *Mobidata - NSF Workshop on Mobile and Wireless Information Systems Workshop Reprot*, May 1995.
- [IK96] T. Imielinski and H. F. Korth. Introduction to Mobile Computing. In *Mobile Computing*, Kluwer Academic Publishers, 1996.

- [Jin94] J. Jing, Weiman Du, A. Elmagarmid and O. Bukhres. Maintaining Consistency of Replicated Data in Multidatabase Systems. In *Proceedings of The 14th International Conference on Distributed Computing Systems*, June 21-24, 1994.
- [JM87] S. Jajodia and D. Mutchler. Dynamic voting. In *Proceedings of the ACM International Conference on Management of Data*, 1987.
- [Joh93] D. B. Johnson. Mobile Host Internetworking Using IP Loose Source Routing. Technical report CMU-CS-93-128 School of Computer Science, Carnegie Mellon University, Pittsburgh, 1993.
- [KH89] W. Kim and F. H. Lochovsky, editors. *Object-Oriented Concepts, Databases and Applications*, ACM Press / Addison-Wesley, 1989.
- [KJ90] B. Kogan and S. Jajodia. Concurrency Control in Multilevel-Secure Databases Based on Replicated Architecture. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, May 1990.
- [KPE92] E. Kühn, F. Puntigam and A. K. Elmagarmid. Multidatabase Transaction Processing and Query Processing in Logic. In Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*, Morgan Kaufmann Publishers, Inc., 1992.
- [KR81] H. Kung and J. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2), 1981.

- [KS92] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.
- [KS93] P. Kumar and M. Satyanarayanan. Log-based Directory Resolution in the Coda File System. In *Proceeding of the 2nd International Conference on Parallel and Distributed Information Systems*, San Diego, CA, January 1993.
- [KS94] H. F. Korth and G. Speegle. Formal Aspects of Concurrency Control in Long-Duration Transaction Systems Using the NT/PV Model. *ACM transactions on Database Systems*, 19(3), 1994.
- [KS95] P. Kumar and M. Satyanarayanan. Flexible and Sage Resolution of File Conflicts. In *Proceedings of the USENIX Winter 1995 Conference*, New Orleans, LA, January 1995.
- [Lai95a] S. J. Lai, A. Zaslavsky, G. P. Martin and L. H. Yeo. Cost Efficient Adaptive Protocol with Buffering for Advanced Mobile Database Applications. In *Proceedings of the 4th International Conference on Database Systems for Advanced Applications, DASFAA '95*, April 1995.
- [Lai95b] J. Lai, B. Mitleman, A. Zaslavsky, P. Granville, A. Rasheed, M. Nolan, M. Faiz and X. D. Zhou. Application Oriented Simulation of a Mobile Distributed Computing Environment. Department of Computer Technology, Melbourne, Technical report CT 95-18, 1995.
- [Lam86] B. Lampson. Designing a Global Name Service. In *Proceedings of the ACM Fifth Symposium on Principles of Distributed Computing, SIGACT-SIGPOS*, Vancouver, CA, August 1986.

- [Lit87] W. Litwin, A. Abdellatif, B. Nicolas, Ph. Vigier and A. Zerounal. MSOL: A Multidatabase Manipulation Language. *Information Science, An International Journal*, Special Issue on DBS, June 1987.
- [LL86] B. Liskov and R. Ladin. Highly-Available Distributed Services and Fault-Tolerant Distributed Garbage Collection. In *Proceedings of the ACM Fifth Symposium on Principles of Distributed Computing, SIGACT-SIGPOS*, pages 29-39, Vancouver, CA August 1986.
- [LM86] C. Lin and D. C. Marinescu. Application of Modified Predicate Transaction Nets to Modelling and Simulation of Communication Protocols. Technical report CSD-TR-599, Purdue University, May 1986.
- [LMR90] W. Litwin, L. Mark and N. Roussopoulos. Interoperability of Multiple Autonomous Databases. *ACM Computing Surveys*, 22(3), September 1990.
- [Lu96] Q. Lu. Improving Data Consistency for Mobile File Access Using Isolation-Only Transactions. Phd Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1996.
- [LW89] P. Lyngbaek and K. Wilkinson. An Overview of the Iris Kernel Architecture. *Object-Oriented Programming Systems*, Pitman Publishing, London, 1989.
- [Lyn83] N. Lybch. Multilevel Atomicity: A New Correctness Criterion for Database Concurrency Control. *ACM Transactions on Database Systems*, 8(4), December 1983.

- [LZ88] W. Litwin and A. Zeroual. Advances in Multidatabase Systems. In *Proceedings of European Teleinformatics Conference on Research into Networks and Distributed Applications*, North-Holland, R. Speth (ed.), April 1988.
- [LZM95] S. J. Lai, A. Zaslavsky and G. P. Martin. A Simulation Model of Adaptive Protocols in Distributed Computing Systems with Mobile Hosts. *Presented at the 28th Annual Simulation Symposium*, Arizona, Phoenix, April 1995.
- [MB90] F. Manola and A. Buchmann. A Functional/Relational Object-Oriented Model for Distributed Object Management, Preliminary Description. Technical Memorandum TM-0149-06-89-165, GTE Laboratories Incorporated, December 1990.
- [Man88] F. Manola. Distributed Object Management Technology. Technical Memorandum TM-0014-06-88-165, GTE Laboratories Incorporated, June 1988.
- [Man89a] F. Manola. Object Model Capabilities for Distributed Object Management. Technical Memorandum TM-0149-06-89-165, GTE Laboratories Incorporated, June 1989.
- [Man89b] F. Manola. An Evaluation of Object-Oriented DBMS Developments. Technical Memorandum TR-0066-10-89-165, GTE Laboratories Incorporated, October 1989.
- [MDR93] B. Marsh, F. Douglis and R. Caceres. Systems Issues In Mobile Computing. Technical Report MITL-TR-50-93, Feb. 1993.

- [Mos85] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, Massachusetts, 1985.
- [MP93] B. Martin and C. H. Pedersen. Long-Lived Concurrent Activities. In M.T. Özsu, U. Dayal and P. Valduriez, editors. *Distributed Object Management*. Morgan Kaufmann Publishers, 1994.
- [MS87] H. Garcia-Molina and K. Salem. SAGAS. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1987.
- [MW82] J. H. Minoura and G. Wiederhold. Resilient Extended True-Copy Token Scheme for a Distributed Database System. In *IEEE Transactions on Software Engineering*, May 1982.
- [Nak93] T Nakajima. Commutativity Based Concurrency Control and Recovery for Multiversion Objects. In M.T. Özsu, U. Dayal and P. Valduriez, editors. *Distributed Object Management*. Morgan Kaufmann Publishers, 1994.
- [NZ84] M. Nodine and S. Zdonik. Cooperative Transaction Hierarchies: A Transaction Model to Support Design Applications. In *Proceedings of the International Conference on Very Large Data Bases*, pages 83-94, 1984.
- [ÖDV93] M.T. Özsu, U. Dayal and P. Valduriez, editors. *Distributed Object Management*. Morgan Kaufmann Publishers, 1994.
- [ÖV91a] M.T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1991.
- [ÖV91b] M. T. Özsu and P. Valduriez. *Distributed Database Systems: Where are we now?* *Computer*, 24(8), 1991.

- [Pap86] C. Papdimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.
- [PB93] E. Pitoura and B. Bhargava. *Dealing with Mobility: Issues and Research Challenges*. Technical Report CSD-TR-93-070, Department of Computer Science, Purdue University, Indiana, 1993.
- [PB94] E. Pitoura and B. Bhargava. *Revising Transaction Concepts for Mobile Computing*, *Proceedings of the 1st IEEE Workshop on Mobile Computing Systems and Applications*, December 1994.
- [PB95] E. Pitoura and B. Bhargava. *Maintaining Consistency of Data in Mobile Distributed Environments*. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, Vancouver, British Columbia, Canada, May 1995.
- [Pit96] E. Pitoura. *A Replication Schema to Support Weak Connectivity in Mobile Information Systems*. In *Proceedings of the 7th International Conference on Database and Expert Systems Applications (DEXA)*, September 1996.
- [PKH88] C. Pu, G. Kaiser and N. Hutchinson. *Split-Transactions for Open-Ended Activities*. In *Proceedings of the 14th International Conference on VLDB*, Los Angeles, 1988.
- [PL90] C. Pu and A. Leff. *Epsilon-Serializability*. Technical report CUCS-054-90 Department of Computer Science, Columbia University, New York, 1990.

- [PL91a] C. Pu and A. Leff. Replica Control in Distributed Systems: An Asynchronous Approach. Technical Report CUCS-053-90, Columbia University, New York, January 1991.
- [PL91b] C. Pu and A. Leff. Epsilon Serialisability. Technical Report CUCS-054-90, Columbia University, New York, January 1991.
- [PS98] E. Pitoura and G. Samaras. Data Management for Mobile Computing. Kluwer Academic Publishers, 1998.
- [RC94] K. Ramamritham and P. K. Chrysanthis. In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness Properties. In M.T. Özsu, U. Dayal and P. Valdureiz, editors. Distributed Object Management. Morgan Kaufmann Publishers, 1994.
- [Reu89] A. Reuter. Contract: A means for extending control beyond transaction boundaries. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, September 1989.
- [RKC92] M. Rusinkiewicz, P. Krychniak and A. Cichocki. Towards a Model for Multidatabase Transactions. *International Journal of Intelligence and Cooperative Information Systems*, 1(3), September 1992.
- [RSG93] K. Ravindran, G. Singh and P. Gupta. Reconfiguration of Spanning Trees in Networks in the Presence of Node Failures. In *Proceedings of 13th International Conference on Distributed Systems*, Pittsburgh, Pennsylvania, May 1993.

- [Rus90] M. Rusinkiewicz, A. Elmagarmid, Y. Leu and W. Litwin. Extending the Transaction Model to Capture More Meaning. *SIGMOD Record*, 19, 1990.
- [RW91] A. Reuter and H. Wachter. The Contract Model. *IEEE data Engineering Bulletin*, 14(1), March 1991.
- [RZ95] A. Rasheed and A. Zaslavsky. Ensuring Database Availability in Dynamically Changing Mobile Computing Environment. Department of Computer Technology, Melbourne, Technical report CT 95-19, 1995.
- [RZ96a] A. Rasheed and A. Zaslavsky. Ensuring Database Availability in Dynamically Changing Mobile Computing Environment. In Proceedings of the 7th Australian Database Conference, Australia, pages 100-108, January 1996.
- [RZ96b] A. Rasheed and A. Zaslavsky. Transaction Management Models in Distributed Mobile Computing Environment. In Proceedings of the 1st MCDA workshop, Australia, February 1996.
- [RZ97] A. Rasheed and A. Zaslavsky. A Transaction Model to support Disconnected Operation in a Mobile Computing Environment. In *OOIS' 97, 4th International Conference on Object-Oriented Information Systems*, Springer-Verlag, New York, USA, 1997.
- [RZ98] A. Rasheed and A. Zaslavsky. Twin-Transactions – Delayed Transaction Synchronisation Model. In *Proceedings of 3rd Workshop on Mobility and Replication*, Brussels, Belgium, July 1998.

- [SBN84] M. Schroeder, A. Birrell and R. Needham. Experience with Grapevine. *ACM Transactions on Computer Systems*, 2(1), February 1984.
- [Sch93] Friedemann Schwenkreis. APRICOTS - A Prototype Implementation of a ConTract System: Management of the Control Flow and the Communication System. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, 1993.
- [SD91] B. N. Schilit and D. Duchamp. Adaptive Remote Paging for Mobile Computers. Technical report CUCS-004-91 Department of Computer Science, Columbia University, New York, 1991.
- [SL90] A. P. Sheth and J. A. Larson. Federated Database Systems for Managing Distributed, Hetrogeneous and Autonomous Databases. *ACM Computing Surveys*, 22(3), September 1990.
- [Son88] S. Son. Replicated data management in distributed database systems. *ACM SIGMOD Record*, 17(4), December 1988.
- [SR86] M. Stonebraker and L. Rowe. The Design of POSTGRES. In *Proceedings of 1986 ACM-SIGMOD International Conference on Management of Data*, ACM, New York, 1986.
- [Sto79] M. Stonebraker. Concurrency Control and Consistency of Multiple Copies of Data in Distributed Ingres. *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 3, March 1979.

- [TD92] C. Tait and D. Duchamp. An Efficient Variable-Consistency Replicated File Service. In *Usenix File System Workshop Proceedings*, Michigan, May 1992.
- [Ter94] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer and B. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the International Conference on Parallel and Distributed Systems*, September 1994.
- [Ter95] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.
- [Tho79] R. Thomas. A Majority Consensus Approach to Cocurrency Control for Multiple Copy Databases. *ACM TODS*, 3(3): 180-209, June 1979.
- [VEH92] J. Veijalainen, F. Eliassen and B. Holtkamp. The S-Transaction Model. In Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*, Morgan Kaufmann Publishers, Inc., 1992.
- [WB84] G. Wu and A. Bernstein. Efficient Solutions to the Replicated Log and Dictionary Problems. In *Proceedings of the ACM Third Symposium on Principles of Distributed Computing*, August 1984.
- [WC96] J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers, 1996.

- [Wei88a] W. Weihl. Commutativity-Based Concurrency Control for Abstract Data Types. *IEEE Transactions on Computers*, 37(12), December 1988.
- [Wei88b] D. Weinreb. An Object-Oriented Database System to Support an Integrated Programming Environment. *Data Engineering*, 11(2), June 1988.
- [Wei89] W. Weihl. Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 11(4), 1989.
- [Wei91] G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. In *ACM Transactions on Database Systems*, 16(1), March 1991.
- [WR92] H. Wächter and A. Reuter. The Contract Model. In Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*, Morgan Kaufmann Publishers, Inc., 1992.
- [WS92] G. Weikum and H. J. Scheck. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In Ahmed K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*, Morgan Kaufmann Publishers, Inc., 1992.
- [YZ94a] L. H. Yeo and A. Zaslavsky. Submission of Transactions from Mobile Workstations in a Cooperative Multidatabase Processing Environment. In *Proceedings of the 14th IEEE CS International Conference on Distributed Computing Systems*, Poland, June 1994.

- [YZ94b] L. H. Yeo and A. Zaslavsky. A Conceptual Framework for Managing Transactions in Multidatabase Environment with Mobile Constituents. Technical Report 94-20, School of Computing & Information Technology, Peninsula, Monash University, Melbourne, Australia, 1994.
- [YZ94c] L. H. Yeo and A. Zaslavsky. An Extended Transaction Model for Multidatabase Systems with Mobile Participants. Technical Report 94-16, School of Computing & Information Technology, Peninsula, Monash University, Melbourne, Australia, 1994.
- [YZ94d] L. H. Yeo and A. Zaslavsky. Layered Approach to Transaction Management in Multidatabase Systems. In *Proceedings of the 5th International Hong Kong CS Database Workshop: Next Generation Database Systems*, 1994.
- [YZ94e] L. H. Yeo and A. Zaslavsky. Simulation Model for Managing Mobile Workstations in Distributed Computing Environment. In *Proceedings of the Australian Telecommunication Networks and Applications Conference*, Melbourne, December 1994.
- [Zas96a] A. Zaslavsky, M. Faiz, B. Srinivasen, A. Rasheed and S. J. Lai. Primary Copy Method and its Modifications for Database Replication in Distributed Mobile Computing Environment. In *15th Symposium on Reliable Distributed Systems*, Canada, October 1996.

- [Zas96b] A Zaslavsky, M. Faiz, B. Srinivasan, A. Rasheed and S. J. Lai. Primary Copy Method and its Modifications for Database Replication in Distributed Mobile Computing Environment. Department of Computer Technology Research Report, Monash University, Melbourne, Technical report CT 96/04, 1996.
- [Zho97] X. D. Zhou, A. Zaslavsky, A. Rasheed and R. Price. Efficient Object-Oriented Query Optimisation in Mobile Computing Environment. In *Technology of Object-Oriented Languages and Systems (TOOLS Pacific 97)*, Melbourne, Australia, November 1997.
- [Zho98] X.D. Zhou, A. Zaslavsky, A. Rasheed and R. Price. Efficient Object-Oriented Query Optimisation in Mobile Computing Environment. *Australian Computer Journal*, 30(2), May 1998.
- [ZM89] S.B. Zdonik and David Maier, editors. Readings in Object-Oriented Database Systems, Morgan-Kaufmann, 1989.
- [ZR96] A. Zaslavsky and A. Rasheed. Efficient Management of Data and Transaction Replicas in Distributed Mobile Computing Systems. *2nd International Baltic Workshop on Databases and Information Systems*, Institute of Cybernetics Estonia, Tallin Estonia, 1996.
- [ZZ97] W. Zhou and J. Zhong. Managing Transactions in a Mobile and Replicated Environment. In Proceedings of the 1st MCDA workshop, Australia, February 1996.

Glossary of Terms

ACID

Traditionally, transactions are expected to satisfy the following four conditions, known as *ACID*ity properties:

- *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.
- *Consistency* requires a transaction to be correct, ie., if executed alone, the transaction takes the database from one consistent state to another consistent state. When transactions are executed concurrently, the database management system must ensure that the execution of a set of concurrent and correct transactions also maintains the consistency of the database.
- *Isolation* requires each transaction to observe a consistent database, ie, not to read the intermediate results of other transactions.
- *Durability* requires the results of a committed transaction to be made permanent in the database in spite of failures.

Concurrency Control

Controlling the interactions of concurrent transactions so that they do not corrupt effects of each other while still executing. It is the activity of coordinating concurrent access to a database in a multi-user database management system. Concurrency control gives the illusion that each user is executing alone on a dedicated system.

Recovery

Recovery basically deals with failures that may prevent in-progress transactions from completion or remove effects of ones already completed. Both cause inconsistencies due to violation of *atomicity* and *durability* properties of transactions. A good system should be able to recover from most type of failures without human intervention.

DBMS

DataBase Management System.

DDBMS

Distributed DataBase Management System.

MWS

Mobile WorkStation.

Disconnected Operation

The essence of disconnected operation is to enable a disconnected host to act as temporary server and continue to service the user/application requirements using the (cache or replicated) copies of the requested data.

Replication

Replication means that some data items are intentionally stored redundantly at multiple sites.

MTM

Mobile Transaction Manager.

FTM

Fixed Transaction Manager.

TTM

Twin-Transaction Manager.

LS

Locally Serialisable.

G1SR

Global One-Copy Serialisability.

GC

Global Certificate.

ASCR

Application Specific Conflict Resolution.

OCC

Optimistic Concurrency Control.

PG

Precedence Graph.

Resynchronise time range Δt

A *resynchronise-time-range* (Δt) is associated with each TTM. A transaction T executed on a TTM at time t must be resynchronised with FTM in time interval $[t, t + \Delta t]$. If resynchronisation is done after this time, the success probability of transaction will decrease.

2PL

2-Phase Locking.

WFG

Wait-for-Graph.

PCD

Probabilistic Conflict Detection.

APCD

Automatic probabilistic conflict detection.

Obsolete transaction

A transaction is *obsolete* if the net effects created by that transaction are either eliminated or made obsolete by later transactions.

Covered transaction

A transaction is *covered* if the removal of a transaction from local transaction history will not affect the resynchronisation process of any other pending transaction. This can be guaranteed if transaction has no *offspring relation* with any other pending transaction.

Off-spring relation

Two transactions T_1 and T_2 have an *offspring relation* if T_2 cannot be re-executed if effects of T_1 are removed.

PRL

Pattern Recogniser and Learner.

Twinning Process

The process of creating two twins (α and β) of a transaction. This is done before starting execution of the transaction.

