Multi-Version Timestamp Certification for Disconnection Protocol in Mobile Computing

by

Yiu Leung Lee

A thesis

presented to the University of Manitoba

in partial fulfilment of the

requirements for the degree of

Master of Science

in

Computer Science

Winnpeg, Manitoba, Canada, 1998

March 25, 1998

MULTI-VERSION TIMESTAMP CERTIFICATION FOR DISCONNECTION PROTOCOL IN MOBILE COMPUTING

BY

YIU LEUNG LEE

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Acknowledgement

First of all, I must thank you my supervisor Professor Ken Barker for guiding me through this thesis. He inspired me in many ways so I could come up with the ideas. My thanks also go to my family for supporting me in Hong Kong, Lan Jo for food and care-taking, and Kevin for proof-reading. Without them, I wouldn't have made this thesis possible. Thank you.

Abstract

Wireless technology inspires many new thoughts to the traditional distributed database systems. "Disconnection" is a standard operation of the mobile computing which the traditional distributed systems usually treat as a failure. A mobile unit informs the fixed servers prior to disconnection. During the time of disconnection, the mobile unit operates by itself with no network support. Upon reconnection, the fixed servers should be able to synchronize the updates and commit the transactions made by the mobile unit during the disconnecting period. Allowing the disconnected mobile units to operate alone may generate some conflicts upon reconnection. This thesis presents the Multi-Version Timestamp Certification for Disconnection Protocol in Mobile Computing to resolve the data inconsistency caused by the disconnection protocol. The objectives of this new protocol includes: (1) to solve data consistency problem between the distributed static servers and moving mobile units and (2) to maximize transaction concurrency for transactions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Rapid technological improvements in wireless communication and the increased functionalities found on small portable notebook computers opens new research opportunities in distributed systems [BAI93, Duc92, IB93, IB94, PB93]. A goal of mobile computing is to provide the greatest mobility possible to the users [BAI93, Duc92, IB93, IB94]. Mobile units are able to move between different locations while remaining *connected* to a wireless network. Mobile computers frequently operate in a disconnected or *doze* mode[1]. Thus, mobile computing is a dynamic distributed system in contrast to traditional distributed systems which are considered static.

Powerful light weight laptop computers have become commonplace recently [HH94, IB93, IB94]. Modern laptop computers can provide the same functionality as desktop computers and, at the same time, provide mobility to users. Users can communicate with LAN, WAN, and Internet through a laptop computer. The large disks now found on laptop computers are able to run *office* sized applications such as word processing and mail program without any network support [BAI93, HH93]. We define office sized applications as those that do not require network support to operate. However, a mobile unit does not work as a file server for both security and availability reasons.

[1] Energy saving mode.

In a sense, a mobile unit is a moving client in client-server environment. Thus, a mobile client functions exactly as any *client*, and it can move and operate in partly or fully disconnected modes.

Mobile computing has created a new application area for existing distributed systems. Several vertical applications[2] of mobile wireless computing including: taxi dispatch, mail tracking, car alarm systems, etc. The most frequently mentioned applications for horizontal applications are *mail enabled* and *information* services to mobile users.

Mail enabled services allow mobile users to send or receive *electronic mails*. Another common application of mail enabled services is *electronic news services*. Electronic news services can deliver current information to mobile users based on individual profiles or preferences. For example: a stock broker may want to know the current status in the stock market while traveling to meet a customer. The broker can connect via a notebook computer and a cellular phone to retrieve the appropriate information.

1.1 Architecture

Two mobility models exist in the current research papers. The first model consists of a fully dynamic environment where everything is mobile in the system [Chr93]. Thus clients and file servers are expected to change locations from time to time. The second model consists of two distinct sets of entities: fixed hosts and mobile units. Some fixed hosts (called *mobile support stations* (MSSs)) are augmented with wireless interfaces to enable communication with mobile hosts and are located within a coverage unit called a *cell*. A mobile unit can then move within a cell or between

[2] *Horizontal* applications are domain independent, as opposed to the *vertical* applications which are written for a specific application domain [BAI93]

two cells while retaining its network connection. Fixed hosts are statically connected by wires and maintain a high-speed connection all the time. The second model is more realistic so most research has focused on that paradigm. Figure 1.1 shows the second model [EJB95, IB94, JBE95, KS92, PB93, PB94a].

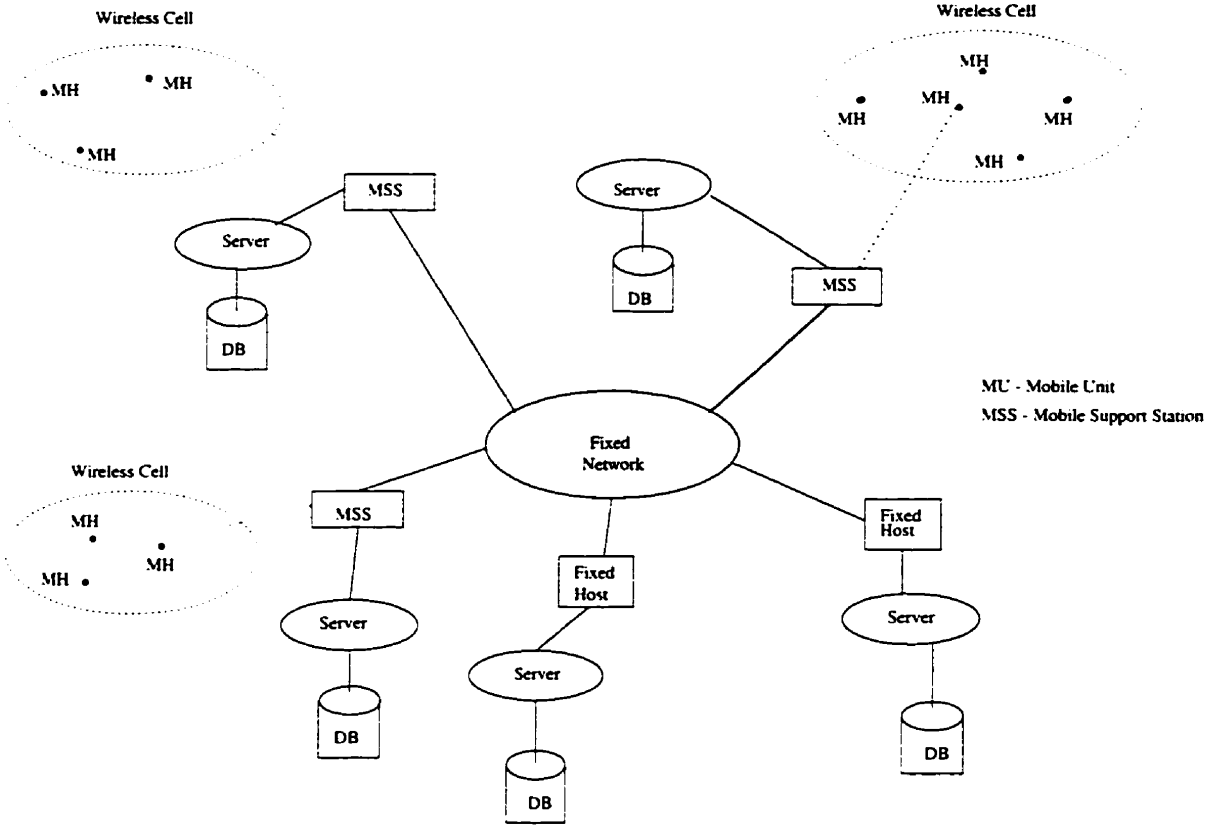


Figure 1.1: Model of a System to Support Mobility

1.2 Operation Modes

For traditional distributed systems, a host operates either in *connected* or *disconnected* mode. Disconnection may be caused by either network failure or server failure. Mobile environment has additional operating modes not typically found on fixed

networks. Pitoura and Bharagava [PB93, PB94a] summarized the different modes, (see Figure 1.2).

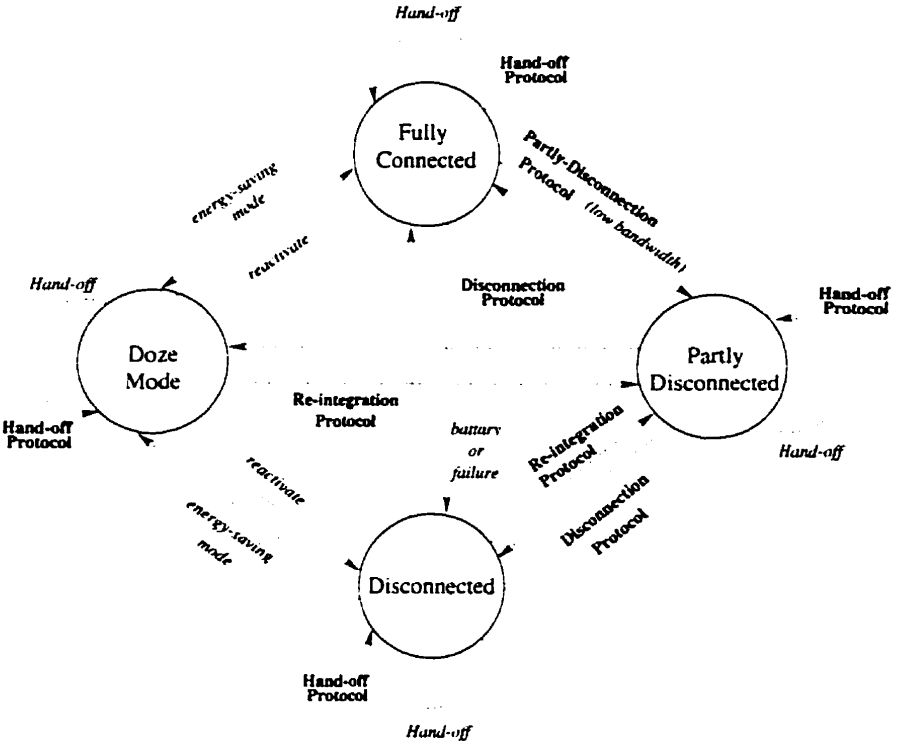


Figure 1.2: States of Operation of a Mobile Unit

In Figure 1.2, there are four states representing the different modes in which a mobile unit can operate. The mobile unit switches its operation mode between them depending on need. The transactions in Figure 1.2 represent the conditions and protocols of switching modes. For example: if a mobile unit is disconnected from a fixed network, it will switch from *fully connected mode* to *disconnected mode* by executing a *disconnection protocol*. After a while, when the disconnected mobile unit wants to save battery life, it switches from *disconnected mode* to *doze mode*. Subsequently, when the disconnected mobile unit wants to reconnect to the fixed servers, it is reactivated back from *doze mode* to *disconnected mode*, then it executes

a *re-integration protocol* and switches from *disconnected mode* to *connected mode*.

When switching modes, the mobile unit is required to execute some protocols. These protocols are used to maintain the smooth transition from one mode to another. For example: when a mobile unit wants to disconnect from the fixed network, the *disconnection protocol* will operate and download all the necessary files and data to the mobile cache. While disconnecting, the *disconnection protocol* keeps running and monitors the transactions running on the disconnected mobile unit. The conditions and responsibilities of the protocols are as follows:

- A *hand-off protocol* is used when the mobile unit wants to cross the boundaries of a cell.

- A *partly-disconnection protocol* is executed when very limited network services are available. A mobile unit should restrict communication as much as possible to the fixed network.

- A *disconnection protocol* is executed before physically detaching the mobile unit from the fixed servers. The disconnected mobile unit can then continue to work using the data in its cache. When the unit reconnects to the fixed servers, the updates made while it is disconnected is then passed to an agent that re-integrates the updates to the fixed servers.

- A *re-integration protocol* is executed when a disconnected mobile unit wants to reconnect to the fixed servers. It helps the system to verify and merge the transactions run on the disconnected mobile unit to the fixed servers. If conflicts are identified, the *re-integration protocol* will attempt to resolve the conflicts.

Although computational power improves rapidly, energy technology improves more slowly. Battery consumption is still a major problem for mobile units [IB93, IB94].

Doze mode operation reduces energy waste on notebook computers because it avoids transmissions until absolutely necessary.

A mobile unit cannot maintain full network connectivity as it must function in *Partly Connected* or *Disconnected* modes. *Partly-Disconnection* and *Disconnection* protocols are developed to handle these situations. The key idea of these two protocol is allowing the mobile unit to continue working while isolated.

1.3 Problem Domain

Several issues differentiate mobile systems from traditional distributed systems. These include:

1. Mobility during the course of a transaction [Chr93, EJB95, GBH96, HH95, IB93, IB94, JBE95, PB93, PB94a, PB94b, PB95].
2. Communication speeds are slower with mobile units [BAI93, IB93, IB94].
3. Transactions are typically long-lived [Chr93, PB93].
4. Maintaining consistency of data over all distributed sites is extremely difficult in a mobile computing environment [HH95, LS94, PB94a, PB94b].

These paradigms introduce new technical issues in the area of distributed database systems [BAI93, Duc92, HH94, HH95, IB93, IB94, PB93]. For example: when a site cannot be reached, traditional distributed database management systems assume that the site is failed. In contrast, the disconnection protocol [Hei92, HH93, HH94, HH95, KS92] is a basic function of mobile computing, so mobile database management systems do not make the same failure assumption. For example: Unit A copies all the necessary files to its cache, and then disconnects from the main office to travel to a branch office. During the journey, a file on the laptop computer is updated. At the

same time, Unit B updates the same file on the fixed server. When Unit A reconnects to the fixed network, a conflict occurs. This scenario does not happen in traditional distributed systems because the *disconnected* unit would be considered *failed.*

Currently, almost all researches concentrate on *Mobility and Scale, Location Management, Bandwidth and Energy Management, Disconnection Protocol, Mobile File Systems* [TD91, ZD93] and *Mobile Transaction Management* [Chr93, EJB95, JBE95, LS94, PB95]. However, very little work has been done on concurrent data access and its impact when disconnection protocol is permitted. The disconnection protocol must be investigated to revise the distributed transaction management algorithms. Figure 1.3 provides an overview of this thesis's goal.

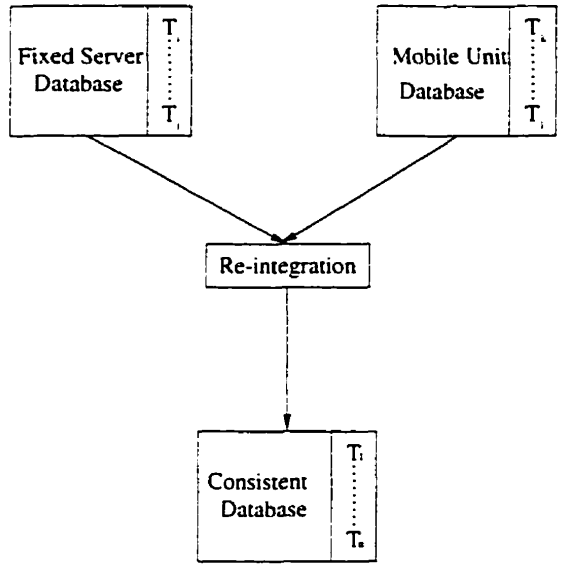


Figure 1.3: General Model of Re-integration

In Figure 1.3, there are four components: (1) a set of transactions running on a fixed server, (2) a set of transactions running on a disconnected mobile unit, (3) a re-integration agent, and (4) a set of transactions running on a consistent database. When a mobile unit is disconnected from a fixed server, it continues its work based

upon the data in its cache, so we have a set of transactions running on the disconnected mobile unit. Concurrently, the fixed server executes its normal daily transactions which creates another set of transactions. When the disconnected mobile unit wants to reconnect, we need a re-integration agent to verify and merge those transactions running on the mobile unit to the fixed server. After the re-integration, the system should return a consistent database. The re-integration agent is the heart of the whole process. It should be able to identify and resolve any potential conflict and guarantee serialization after merging the transactions on the mobile unit to the fixed server.

1.4 Motivations

Many modern proposed solutions to the conflict problem involve some user input [Hei92, HH93, HH94, KS92, LS94]. For example: suppose that a travel company downloads all the information such as airlines' schedules and hotels' reservation once a day. All the travel agents in the company share the data locally instead of connecting to different networks. In this particular case, all of the travel agents are working with "disconnected data". Suppose that an agent helped a customer to schedule a vacation in Miami. The agent asked the customer his favourite airlines, hotel and local tours he wants to join in Miami. The customer might choose to take United Airlines to Miami, stay in Holiday Inn and join a local tour to Disney World. For this transaction, there are three operations: $op1$: book an air-ticket, $op2$: reserve a hotel room, and $op3$: reserve a space in the tour to Disney World. Unfortunately, when the system reconnected back to the server, all seats from United Airlines have been already booked so the transaction must be aborted. This thesis argues that the agent could submit one more *alternative* transaction. If the master transaction contains conflicts, the alternative transaction will substitute it. In this example, the master

transaction is to book a ticket from United Airlines; the alternative transaction could be to book a ticket from American Airlines. This thesis shows that the transaction manager can automatically replace the conflicting transaction using the alternative transaction and commit it.

Furthermore, during the time of disconnection, the database showed 100 empty rooms left in Holiday Inn. However, the Holiday Inn's database shows that only 90 rooms are left when connected. In the agent's point of view, 90 or 100 empty rooms make no difference as long as he can reserve a room for his customer. In the transaction management's point of view, $op2$ violates the serializability because it read stale data. Traditional approaches will abort and roll back this transaction even though only $op2$ generates conflict. But, this thesis argues that using static analysis [GB95], we can avoid re-executing the whole transaction.

Let us consider another case: A medical company owns two hospitals. In order to reduce operational expenses, only one accountant is employed so he is responsible for both hospitals. The accountant spends only one day in each hospital monthly, and the rest of the month in the headquarter. Each time he travels to one of the hospitals, he downloads the financial information from the hospital's database in his notebook and audit it in the headquarter. In this case, the accountant is working with a "disconnected data". In the end of each month, the accountant travels back to the hospitals, reconnects his notebook and submits the transactions he did in the headquarter back to the hospital's database. Since the information such as employees' salaries and maintenance fee does not change frequently, we can expect that most of the transactions can be committed successfully. But, suppose that the management of the medical company agreed to increase one of the doctor's salary in hospital A, this change does not reflect on the accountant's notebook immediately. If the accountant submits a transaction which pays salaries to all 200 employees in hospital A, this

transaction must be aborted and rolled back even though only one salary is changed. This scenario is inefficient and undesirable.

This thesis argues that previous approaches need to re-execute the whole conflicting transaction which is not always necessary [PB95]. A *Multi-version Timestamp Certification for Disconnection Protocol in Mobile Computing* is presented to address the new transaction paradigms found in mobile computing environment. The primary objectives of the algorithm are:

1. solving data consistency problem between the distributed static servers and moving mobile units, and

2. maximizing transaction concurrency for transactions.

Before explicitly discussing our approach, the *Multi-Version Timestamp Ordering Algorithm* and *Optimistic Certification Scheme* are required. Our model is based upon these two algorithms.

1.4.1 Multi-Version Timestamp Ordering Algorithm

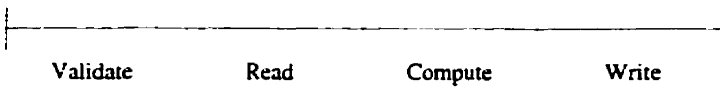
Timestamp Ordering algorithm (TO) [BG80, OV90] ensures serializability using a non-atomically increasing logical timestamp. A basic TO algorithm never causes transactions to wait, but may require them to start. To restart transactions is undesirable, so *multi-version timestamp ordering* algorithm (MVTO) [Pap84, Tho78] attempts to eliminate the restart overhead. MVTO does not modify the database using a *write* operation. Instead, it creates a *new version* of that data item marked by the timestamp. If a *read* operation ($read(x)$) is issued, the system will fetch a *version* of x where $ts(x_v)$ is the largest timestamp less than $ts(read(x))$ to the operation. For example: T_i wants to read x. However, $T_j = \{w_j(x)\}$ has been committed where $ts(T_i) < ts(T_j)$. In basic TO, T_i must be aborted and restarted. But, in MVTO, T_i

can still commit because T_i can read the latest version of x where $ts(x_v) < ts(T_i)$. The MVTO trades space for time because it requires extra storage to keep the old version of data.

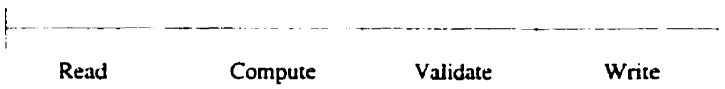
1.4.2 Optimistic Certification Scheme

Kung and Robinson [KR81] designed an optimistic concurrency control algorithm for a centralized database of low concurrency. Sinha, Nanadikar and Mehndiratta [SNM85] moved one step further and introduced a timestamp based certification algorithm for distributed database systems. An optimistic concurrency control algorithm assumes that concurrent transactions are infrequent, so it will not check for any conflicting operation until the *End_Transaction* is issued. The algorithm commits the transaction if no conflict occurs, or aborts and restarts it otherwise. Thus, a consistent database is always maintained. This algorithm is more efficient for mostly-read-only database such as query-domain database because there is no locking involved, thereby eliminating the associated delays.

Pushing the validate phase after compute phase reduces the overhead time because the transaction manager allows transactions to compute first without checking any conflict among them. This scheme reduces the time spending on predicting potential conflicts during the execution of transactions. A transaction is validated by the transaction manager when a commit signal is received. If no conflict occurs, the transaction is committed; otherwise, it is aborted. In a lightly-shared system, conflicts are rare. Using optimistic certification scheme can save the overhead time of predicting conflicts. However, if a system shares its files heavily, the optimistic approach is not appropriate because many transactions must be aborted and redone. Figure 1.4 shows the sequence of phases of pessimistic and optimistic transaction executions.



a. Pessimistic Concurrency Protocol



b. Optimistic Concurrency Protocol

Figure 1.4: Phases of Transaction Execution

MVTO is suitable for a mobile computing environment because it does not abort *read* operations which are mostly used in mobile computing environment. A stand-alone disconnected mobile unit is operated in isolation: hence simultaneous sharing of data should happen rarely which suggests the certification scheme be used.

This thesis develops a new transaction model similar to [GBH96]. The new model is able to record the modifications in a transaction at execution time in the disconnected mobile unit and provide information to analyze and resolve conflicts using static analysis [GB95] upon reconnection.

A locally committed mobile transaction will result in a conflict if stale data is read during the time of disconnection. Traditional approaches can only base on the dynamic *read/write* operations or transaction submission order alone to roll back the transaction. Such approaches are based on exact but extremely limited information. Applying the static analysis can derive more extensive but inexact information to resolve the conflict at compile time and guide the transaction manager to commit the transaction at run-time [GB95]. Thus, concurrency control overhead is reduced by shifting some of the effort to compile time.

Unlike run-time scheduling method to detect the conflict at run time and roll back

conflicting transactions, a static analysis retrieves and analyzes the static information of a transaction to predict and resolve conflicts prior to the execution of commit at run time. In other words, a mechanism is developed to analyze the static information of a transaction to predict where conflicts arise and attempt to resolve them prior to committing the transaction. The consequences of using static analysis can remove any overhead associated with scheduling and guarantee a successful commit at run time.

During the execution time, we allow the fixed servers and disconnected mobile units to run their transactions concurrently. Upon reconnection, we apply *Partial Re-execution* [HBG97] using static analysis to analyze and reconcile the conflicting transactions at compile time before committing the transactions at run time.

1.5 Contributions

This thesis attempts to achieve the following:

1. Investigate the properties of a mobile computing environment. Furthermore, identify the problems and difficulties to allow concurrent data access in both the fixed servers and disconnected mobile units.

2. Review the existing research literature and determine their limitations in solving data inconsistency caused by the disconnection protocol in a mobile computing environment.

3. Revise the traditional distributed transaction model and introduce an improved mobile transaction model suitable for the disconnected mobile unit.

4. Combine Sinha, Nanadikar and Mehndirtta's [SNM85] and Hadaegh, Barker and Graham's [HBG97] work to develop a new transaction management for the

disconnection protocol.

5. Implement, demonstrate and analyze the results of the new transaction model. From the results of the experiment, we discuss its performance and limitations.

1.6 Assumptions

Based upon the observation [KS92, SKM+93], the authors conclude that laptop users are very aware of the operations they use while traveling [HH93]. Moreover, they will eventually connect their laptop computers back to the fixed servers and report their updates during the time of disconnection. Furthermore, to prevent the lost of critical data items from the fixed network, we do not allow the moving disconnected unit to hold any primary data. All data in the mobile cache is replicated from the fixed servers. To reduce the overhead time and complexity of re-integration, we do not allow any direct communication between disconnected mobile units. The assumptions are summarized below:

1. Since people are able to operate for extended period in isolation, they are quite good at predicting their needs for future file access.

2. The workload of engineering/office applications generally consists of sequential read-write sharing, but little simultaneous sharing [HH93, SKM+93].

3. The disconnection period is short, and the disconnected mobile unit will eventually reconnect back to the servers [Hei92, HH93, HH94, SKM+93].

4. The data in the fixed servers is the primary one. The data in the disconnected mobile is replicated from the fixed servers.

5. Mobile units do not directly communicate with each other. Any inter-mobile data exchange occurs through connection to the fixed network.

1.7 Thesis Organization

This thesis is organized as follows: Chapter 2 discusses the related work to implement the disconnection protocol. and designing the mobile transaction model and management system. Chapter 3 presents a new mobile transaction model. Chapter 4 presents the general model and overview of the *Multi-Version Timestamp Certification for Disconnection Protocol in Mobile Computing.* The detailed implementation of the *Local Transaction Certification Scheme* will also be presented in Chapter 4. Chapter 5 discusses and presents the overview and detailed implementation of the *Global Commit Protocol.* The demonstration of mobile computing is presented in Chapter 6. Finally, conclusions and future work are given in Chapter 7.

Chapter 2

Related Work

2.1 Introduction

Much work has been done on the mobile computing environment [Chr93, EJB95, GBH96, Hei92, HH93, HH94, JBE95, KS92, LS94, PB95, TD91, ZD93]. We can divide them into two categories: (1) Mobile File Systems and (2) Mobile Transaction Managers. First, we discuss the effects of adding mobility to the distributed file systems and some related work that solves their problems. Then, we discuss the mobile transaction managers and provide some new methods to revise the traditional distributed transaction managers to meet the needs of a mobile transaction manager.

2.2 Mobile File System

Several issues have been addressed when adding mobility to distributed systems [Hei92, HH93, HH95, KS92, TD91, ZD93]. This thesis concentrates on the effect of allowing the *disconnection protocol*. The disconnection protocol in mobile computing is different from failure. The mobile units can inform the fixed network of an impending disconnection prior to its occurrence and execute a disconnection protocol. A

disconnected mobile unit can continue working using data in its cache. When the unit reconnects, it passes the updates made while disconnected to an agent that re-integrates those updates to the servers.

Generally, a mobile unit copies files from a network, works on the files when it is disconnected, and then restores the updated information to the network. The system should prevent the user from accidentally overwriting another user's work in an automatic way [HH93]. Although the strength of the disconnection protocol is it permits mobile units to continue working when the network is inaccessible, inconsistency of the shared data in the disconnected mobile units and servers must be carefully avoided.

2.2.1 Disconnection Protocol in Coda

Kistler and Satyanarayanan [KS92] implement the disconnection protocol in the Coda file system[1]. They implement the disconnection protocol as a user-level process called *Venus*. Venus uses optimistic replication. The assumption is that simultaneous write-sharing happens rarely in the UNIX NFS so an optimistic approach will yield better performance in a lightly-shared system. Venus operates in one of the three states: *hoarding*, *emulation*, and *re-integration*. Figure 2.1 illustrates transaction states found in a Venus process.

Venus is in a *hoarding* state while the mobile unit is connected. It replicates server's data, executes transactions in the mobile unit, and reports updates to the server. Venus moves from a hoarding state to an *emulation* state while it is disconnected. Venus emulates server's operations when the network connection is inaccessible. During the emulation state, the cache manager logs all the accessed objects and reports to the user if a cache miss occurs. During reconnection, Venus enters the

[1]Coda file system is a distributed system developed on UNIX.

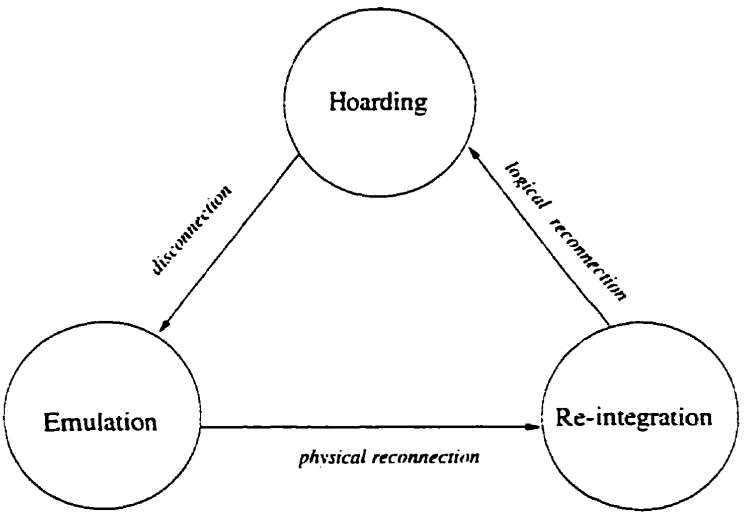


Figure 2.1: Venus States and Transaction

re-integration state, resynchronizes its cache with the server, and returns to hoarding state. In the re-integration state, they propose a *replay algorithm* to re-integrate the data in the local cache and the data in server together.

The replay algorithm consists of four phases. In phase 1, the log in the disconnected mobile unit is examined, a transaction is begun, and all objects referenced in the log are locked. In phase 2, each operation in the log is validated and then executed. The validation contains conflict detection and integrity protection. In phase 3, the actual data transfer process is done. In phase 4, the transaction is committed and all locks are released. Venus's optimistic algorithm means conflict is possible at re-integration. The authors only consider *write/write* conflicts because they claim *read/write* conflicts are irrelevant to the UNIX file system because it has no notion of atomicity beyond the boundary of a single system call. The conflict detection is straightforward. Each data item has its own *storeid* that uniquely identifies the last update to it. During phase 2 of the replay algorithm, a server compares the *storeid* of every data item in the log of the mobile unit with the *storeid* of its own replica of

the data item. If the comparison indicates equality for all data items, the operation is performed and the mutated data items are tagged with a new *storeid* specified in the log entry. If a *storeid* comparison fails, the entire re-integration is aborted.

Venus attempts to provide the disconnection protocol for the Coda file system. It modifies the existing distributed file system and cache manager to guarantee availability and consistency of data items. Unfortunately, it does not automatically solve the problem of data inconsistency caused by conflicting transactions during the time of re-integration. The replay algorithm only logs and reports to the user where conflicts occur. Users are required to resolve conflicts manually.

2.2.2 Disconnection Protocol in AFS

Similar to Venus, Huston and Honeyman [HH93] present the disconnected operation in AFS[2] which also modifies the existing distributed file system and cache manager to make the disconnection protocol possible. The remote cache manager logs all the accessed data items during the time of disconnection, and compares the logs to the file server upon reconnection. The disconnected operation in AFS resolves conflicts by copying the conflicting object in the mobile unit to a new object in the file servers and informs the user. The user then decides which objects he wants to discard. Once again, the disconnected operation in AFS concentrates on the effect of the distributed file system. The transaction management issues are not addressed explicitly.

2.3 Mobile Transaction Management

Adding mobility to the distributed transaction managements creates new problems that need to be researched. There are two major issues: (1) Relocation Problem,

[2]Andrew File System (AFS) is a distributed file system built on UNIX.

and (2) Frequent Disconnection. Much work [GBH96, JBE95, LS94, PB95] has been undertaken. The work focuses on maintaining serializability and minimizing message exchange in the network.

2.3.1 Isolation-Only Transaction for Mobile Computing

Lu and Satyanarayanan [LS94] develop the *Isolation-Only Transactions* (IOT) for mobile computing. An IOT is a flat sequence of file access operations bracketed by a *begin_iot* and an *end_iot*. The IOT execution model uses an optimistic concurrency control mechanism so the client's local cache is a private workspace for transaction processing. There are two classes of transactions for the IOTs: (1) First class transaction whose execution does not contain any partitioned file access, and (2) Second class transaction whose execution contains partitioned file access. Partitioned file access means that a file is shared by both the fixed server and the disconnected mobile unit. A first class transaction's results are visible on the servers once it commits. A committing second class transaction enters the *pending* state so it can be validated later. Figure 2.2 depicts the state transition diagram for IOT execution.

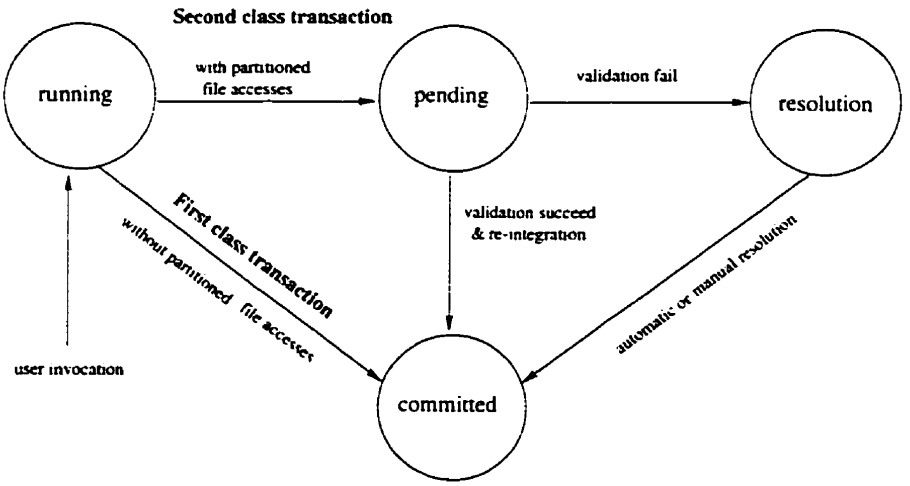


Figure 2.2: A State Transition Diagram for IOT Execution

In the diagram, there are three cases where a transaction can be committed. The IOT execution guarantees that a first class transaction is always serializable because no partitioned data item is accessed. A second class transaction is always serializable locally but may not be serializable globally. If a second class transaction is not in conflict during global validation, the transaction will leave the pending state and be committed globally. If this option fails, the system will suggest 4 options: (1) re-executing the transaction, (2) invoking the transaction's application specific resolver, (3) aborting the transaction, and (4) notifying the users and wait for further decision. For example: if a transaction is used for editing the files of a co-authored paper on a disconnected laptop, this option is useful for coordinating possible concurrent updates. Unlike the disconnection protocols in Coda and AFS, IOT attempts to maintain transaction serializability. However, it cannot resolve conflicts at the system level so the user must decide *what to do*.

2.3.2 Optimistic Two Phase Locking for Mobile Transaction

Jing, Bukhres and Elmagarmid [JBE95] adapt the optimistic two phase locking (O2PL) algorithm to the mobile environment and call it the optimistic two phase locking algorithm for mobile transaction (O2PL-MT). The O2PL algorithm is developed for distributed replicated database systems and uses a read-one write-all concurrency control algorithm. The word *optimistic* is used because *write_locks* are obtained just before the commit phase begins, but *read_locks* are obtained immediately from the local or nearest copy site when the *read* operation is issued. A *read* operation is very inexpensive within its local site when it is compared to the message-intensive approach when site boundaries are crossed.

However, the authors show that mobility results in extra messaging for the O2PL algorithm. For example, suppose that a transaction $T_i = \{r(x)r(y)w(z)\}$ is executed

in a moving mobile unit. So, it issues $r(x_1)$ in site 1, $r(y_2)$ in site 2 and $w(z_3)$ in site 3, and x and y are replicated in sites 1, 2, and 3. When T_i is committed in site 3, it must send $unlock(x_1)$ to site 1 and $unlock(y_2)$ to site 2. Thus, the system sends extra messages. Figure 2.3 depicts the example.

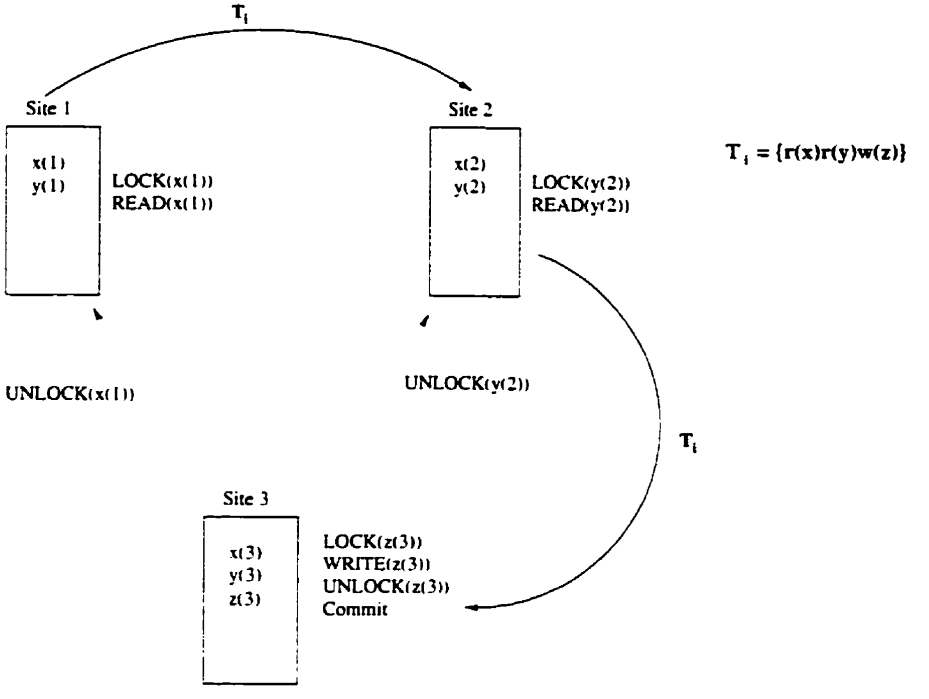


Figure 2.3: A Mobile Transaction Example

In this traditional distributed systems, this scenario will not happen because the positions of transactions are assumed to be fixed. The authors claim that these extra message transmissions can be avoided in mobile computing.

Consider the earlier example. Instead of sending $unlock(x_1)$ to site 1 and $unlock(y_2)$ to site 2, T_i sends $unlock(x_3)$ and $unlock(y_3)$ to site 3 at commit time. The algorithm itself is not sufficient to guarantee the correctness of read-one write-all criteria, so some additional issues have to be explored: (1) *read_locks* must remain at remote sites until the coordinator decides to release the locks and commit a transaction lo-

cally. (2) An update transaction must be able to determine that the item to be updated has not been locked by other transactions for reading if the *read_lock* and *unlock* are executed at different sites. Such a check should have a low message cost. (3) A mechanism much be provided to remove the pending *read_locks* at remote sites at proper time if the continuation of such locks will affect the execution of other transactions.

In the traditional O2PL algorithm, when a transaction requests a *write_lock*, the system will boardcast to all remote sites. If no pending *read_lock* is in effect, the transaction grants the *write_lock*; otherwise, it is blocked. This process requires only one round of message exchange. However, if the *read_lock* and *unlock* are executed in different sites, we need two rounds of message exchange to ensure the read-all write-one approach. In the first round, the system collects *unlock* information from all the copy sites involved. In the second round, while all sites indicate that the *unlock* has been executed, the system can send a message to allow the *write_lock* request to be granted.

The O2PL-MT algorithm is designed for a mostly-read-only mobile system. As we can see, granting a *write_lock* requires two-round of message transmissions so a heavily write-shared system will increase the number of message transmissions. Moreover, it addresses only the mobile relocation problem, it does not support the disconnection protocols explicitly.

2.3.3 Mobile Transaction in Clustering Mobile System

Pitoura and Bhargava [PB94a, PB94b, PB95] describe a transaction model for mobile computing. They partition the mobile system into several clusters which are smaller in size and the clustered data is closely related. Transactions must maintain strict consistency within its cluster, but not necessarily across other clusters. The mobile

unit and fixed servers can issue either *weak transactions* or *strict transactions*. For critical transactions, users issue strict transactions which guarantee global consistency when the transaction commits. When a strict transaction wants to commit globally, the system sends a commit request message to every participating site including mobile units and waits for their replies. This may be quite costly for mobile units because they may not be easily located. Moreover, some mobile units may operate in isolation so the system must either discard the transaction or wait for the mobile units' replies.

On the other hand, a weak transaction only requires strict consistency within its local cluster so it avoids the overhead of long network accesses. The authors argue that data clusters requiring shared access is rare across cluster boundaries. For example: in a university community, the Department of Computer Science does not share its data heavily with the Department of Statistic. Thus computer science users issue weak transactions within its local cluster, as do the statistics users. If a user does not require data consistency across two clusters, it will never release strict transaction.

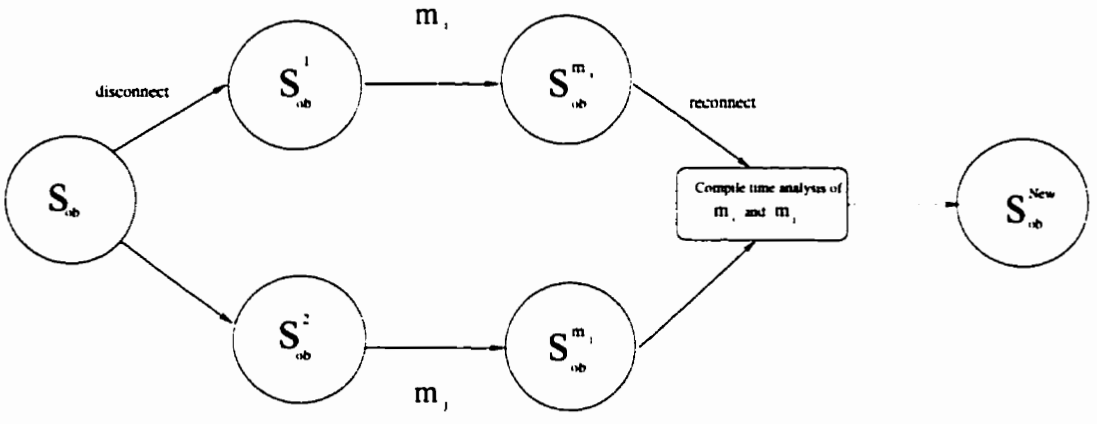
Dividing mobile systems into clusters is an extension of network partitioning. However, two or more separated clusters may eventually need to merge their data. This merging can lead to conflict between the two clusters. A weak transaction issued in a cluster may contain inconsistent data that must be resolved when a strict transaction is issued. Resolution of inter-cluster serializability is accomplished with roll back transactions whose weak writes conflict with strict transactions [PB95]. Care must be taken to correctly merge cascading aborts that occur when the transaction manager re-does a transaction.

2.3.4 Reconciliation in a Nested Object Transaction Environment

Graham, Barker and Reza-Hadaegh [GBH96] extend the disconnection protocol to an object-base system. They argue that the property of encapsulation in objects helps the re-integrating agent to reconcile conflicting transactions. In an object-based system, every object is encapsulated. If an operation wants to access any information in a particular object, it must go through the public protocols of the object. It immediately tells us that the system knows exactly what data items the operation wants to access and what behaviour the transaction expected from the object. Based upon these definitions, a new concept of optimistic re-integration algorithm is described in the paper. A mobile unit downloads all the critical objects from the object-based system prior to disconnection. While the mobile unit is disconnected, the cache manager is responsible for logging all the object's behaviours at execution time. Since every object is encapsulated, the transaction manager knows what and how the changes occur. At the same time, the transaction manager in the fixed file server logs all the modifications to the critical objects. During the reconnection, if there is any conflicting transaction reported, the re-integrating agent analyze the behaviour of the critical objects before executing global commit. It examines the changes logged on both objects in the mobile unit and the server during execution time, and determines whether the changes should be integrated or not. An object *ob* can be replicated. We use S^i_{ob} to represent i replication of *ob*. An object *ob* moves from state S_{ob} to $S^{m_i}_{ob}$ using method m_i as $S_{ob} \xrightarrow{m_i} S^{m_i}_{ob}$. According to the authors [GBH96], S^{m_i,m_j}_{ob} is equal to S^{m_j,m_i}_{ob} so the execution sequence of two methods does not affect the final result. For example: a mobile unit replicated an object S^1_{ob} in its cache. While the mobile unit is disconnected, it updated S^1_{ob} to $S^{m_i}_{ob}$ using m_i. At the same time, the fixed server updated its replica of S^2_{ob} to $S^{m_j}_{ob}$ using m_j. Upon reconnection, the re-integrating

agent merges $S_{ob}^{m_i}$ and $S_{ob}^{m_j}$ to S_{ob}^{new}.

Figures 2.4 depicts the process of object re-integration and illustrates two cases to re-integrate two conflicting objects to a new consistent object. The system can integrate S_{ob}^{1} and S_{ob}^{2} to S_{ob}^{new} in two cases.



where

S_{ob} is the based object.

S_{ob}^{1} is the object cached by the disconnected mobile unit.

S_{ob}^{2} is the same object which resides on the server.

S_{ob}^{1} and S_{ob}^{2} are identical.

m_1 and m_2 are the methods operated in S_{ob}^{1} and S_{ob}^{2} correspondingly.

$S_{ob}^{m_1}$ is the new object in the disconnected mobile unit after performing method m_1

$S_{ob}^{m_2}$ is the new object in the fixed file server after performing method m_2

S_{ob}^{new} is the new object after the re-integration of $S_{ob}^{m_1}$ and $S_{ob}^{m_2}$

Figure 2.4: Reconciliation of Object-Base System

- Case 1:

$$S_{ob}^{new} = S_{ob}^{1} \xrightarrow{m_i} S_{ob}^{m_i} \xrightarrow{m_j} S_{ob}^{m_i.m_j}$$

- Case 2:

$$S_{ob}^{new} = S_{ob}^{2} \xrightarrow{m_j} S_{ob}^{m_j} \xrightarrow{m_i} S_{ob}^{m_j.m_i}$$

S_{ob}^{new} can be obtained either from S_{ob}^{1} with method sequence m_i and m_j or from S_{ob}^{2} with method sequence m_j and m_i. Since the re-integrating agent knows what and how the objects have changed, the replay algorithm reconciles two methods to

produce a new consistent object. The worst case is discarding one of the critical objects and re-executing the conflicting transaction.

The authors present a reconciliation mechanism suitable for a close nested object transaction environment. However, their whole idea is based upon the property of object-orientation. For other database models such as relational and network models, this method is not appropriate.

Chapter 3

Mobile Transaction Model

Transactions are used to ensure consistent and reliable data management in addition to atomic and isolated user interaction. In traditional database systems, a consistent and reliable transaction has to satisfy the ACID properties [OV90]. Recent research [Chr93, HBG97, PB94b] has pointed out that advanced applications employing complex data structures such as CAD and object base systems may require a relaxation of those properties. The invention of wireless medium and the disconnection protocol has changed the concept of transaction management in distributed systems [EJB95, GBH96, HH95, IB93, IB94, JBE95, LS94, PB95]. For example, in the traditional transaction management, disconnection means failure, but the disconnection protocol in mobile computing environment is also a standard operation. This thesis introduces a new mobile transaction model suitable for the disconnection protocol and shows that it is sufficient to maintain correct data. The correctness criterion used in this thesis is conflict-serializability.

3.1 Transaction Properties

Transactions ensure that the database remains consistent even when concurrent accesses and failures occur. In traditional database systems, the consistency and reliability aspects of transactions are due to four properties (ACID):

- *Atomicity* states that all or none of the operations are executed.
- *Consistency* states that a transaction must take the database from one consistent state to another.
- *Isolation* means that no partial result is seen. It guarantees a consistent view of the database at all times.
- *Durability* means that once a transaction commits, its results are permanent.

In mobile computing environment, protecting the ACID properties requires new research. For example: mobile units frequently work in disconnected mode. If we allow concurrent data access in both the fixed server and the disconnected mobile unit, a new mechanism is required to guarantee data consistency. Furthermore, wireless communication is not as stable and, as reliable when compared to wired communication. We should expect more frequent errors in a mobile transaction so a different recovery method is required to guarantee durability. In our model, we develop a new *Multi-Version Timestamp Certification for Disconnection Protocol in Mobile Computing* (MVTC) to ensure consistency optimistically. MVTC consists of two phases: (1) *Local Timestamp Certification Scheme* (LTCS) runs on the disconnected mobile unit and guarantees only serialization among local transactions, and (2) the *Global Commit Protocol* (GCP) ensures global serialization upon reconnection. We use conflict-serializability for correctness. The general idea of the GCP is shown in Figure 3.1.

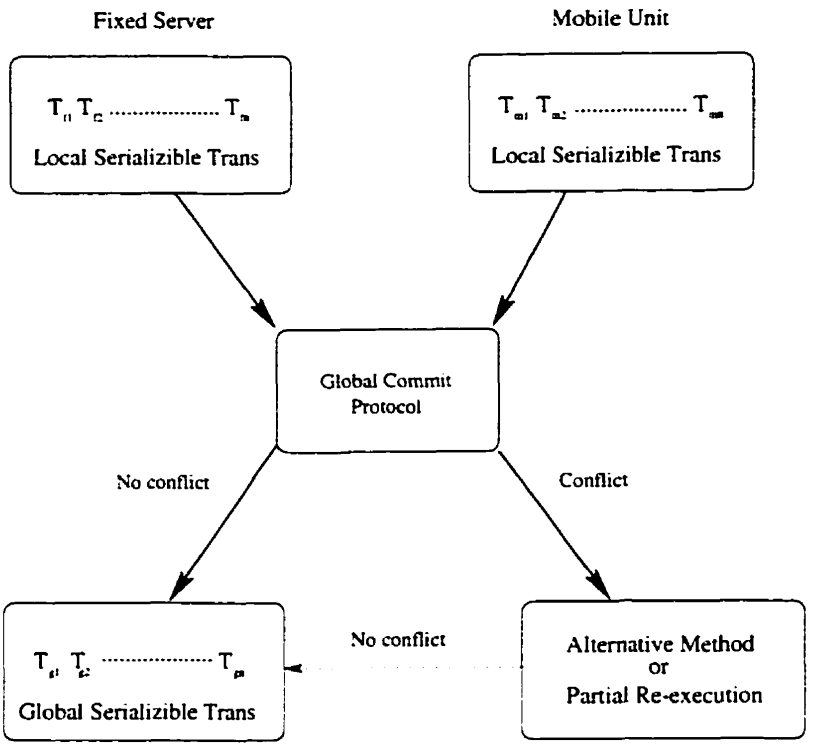


Figure 3.1: Model of MVTC

In Figure 3.1, there are two sets of locally serializable transactions: one set is running on the disconnected mobile unit and the other set is running on the fixed server. Upon reconnection, the GCP will examine every transaction in the disconnected mobile unit and identify any potential conflict. If there is no conflict, the transaction will be committed and merged to the fixed server. Otherwise, the system will attempt to resolve the conflicts. In our model, we provide three methods to resolve conflicts. (1) execute and commit the *alternative transaction*, (2) *partially re-execute* [HBG97] the conflicting transaction, and (3) abort the transaction. After resolving conflicts, the system commits the resolved transaction globally. The detail of these resolutions is presented in Section 3.5.

3.2 Mobile Transaction

A mobile transaction contains several sub-transactions executed on mobile and fixed units. Given the high cost and low bandwidth of mobile communication, users are often willing to temporarily work with '*stale*' data if correctness is guaranteed [HH93, HH94, IB93, IB94, LS94, PB95]. Consider a disconnected mobile unit requesting a hotel reservation. The mobile unit may show 100 rooms are available even though 95 rooms are left based upon data on the fixed servers. Clearly, it makes no difference to the mobile user since only one room is required. In our model, we allow the room reservation to be committed locally and confirm it with the GCP to verify this committed mobile transaction later. Although all 100 rooms may be unavailable when the mobile unit reconnects, this scenario is unlikely to happen for a short disconnection typical in mobile computing. Strictly speaking the transaction has violated serializability because the transaction has read stale data locally.

The GCP examines every locally committed transaction. If there is any conflict, the GCP will either submit an *alternative transaction* or *re-execute* the transaction partially. If the GCP fails to resolve conflicts, it will abort the transaction. Without loss of generality, we assume that at any given time, a mobile unit can directly communicate with only one mobile support station which is responsible for the logical or geographical area in which the mobile unit moves.

3.2.1 Transaction Structure

Traditionally, a transaction is composed of a sequence of *read/write* operations on database items bracketed by **Begin_Transaction** and **End** statements. A **Begin_Transaction** statement results in a call of the transaction manager at the user's site which processes all user's requests following it. This is a *flat* model for transaction with only one **Begin_Transaction** and **End** statement.

Suppose that a stock broker (user A) helps a new client (user B) to invest. First, user A checks the current price, then the credit history of user B, and only if both results are satisfactory will user A help user B place the order. This transaction involves accessing several different databases and requires dependencies between sub-transactions. The transaction manager first divides the transaction into a number of sub-transactions. They run concurrently perhaps on different database systems and report the results back to the transaction manager. This model is a *nested* transaction model. To make the case more complicated, user A may disconnect from the network temporarily and transfer to a new physical location. During the period of disconnection, user A may access and modify some of its cached data items. This thesis concentrates on resolving the serialization of transactions affected by the disconnection protocol.

We could lock all the data items while a mobile unit disconnects from the rest of the network to maintain serialization. However, it is very inefficient and unnecessarily restrictive. Our model allows the disconnected mobile units to operate and commit locally using cached local data items, and commit globally upon reconnection. No locking is involved during the transaction execution in this model. However, when the disconnected mobile unit reconnects to the network, some mechanisms are required to detect and resolve potential conflicts between fixed and mobile units. When conflict occurs, the system can prompt the user to determine if a conflicting transaction needs to be discarded. By providing a reconciliation methology, human interaction is minimized [HH93, HH94, KS92].

3.3 Definition of Conflict

We define an operation r of transaction i as $op_{i,r}$. We define a boolean function called $conflict$ which accepts two operations and returns *true* if at least one of them is a

write operation. We define that $op_{i,r}$ proceeds before $op_{j,s}$ as $op_{i,r} \rightarrow op_{j,s}$. We say that a transaction T_i must be serialized before T_j if

$$\exists op_{i,r}, op_{j,s}(conflict(op_{i,r}, op_{j,s}) \wedge (op_{i,r} \rightarrow op_{j,s}))$$

3.4 Data Consistency

Mobile units are only connected to the fixed network intermittently, so mobile transaction management must reflect increased concern for bandwidth consumption and disconnection constraints. Disconnection constraints are caused if the system allows concurrent access of data between the fixed server and the disconnected mobile unit because conflicts are possible.

Since pessimistic approaches induce unacceptably high costs, this thesis attempts to attain consistency after the mobile unit reconnects to the network. When a local transaction issues **Begin_Transaction** in the disconnected mobile unit, the *local transaction manager* (LTM) is called. The LTM logs all information such as *timestamp* and *flags* data items accessed by the transaction. Upon reconnection, the GCP merges the transactions in the disconnected mobile unit to the fixed server and forms a consistent database.

In our model, transactions are allowed to run concurrently in both the disconnected mobile unit and the fixed servers which results in a substantially higher concurrency than the locking method [KR81, OV90, SNM85]. Transactions are classified into two categories: a *local transaction* is one whose execution guarantees local serializability; a *global transaction* is one whose execution guarantees global serializability. The global transaction is very different from the local transaction. It can only be verified for global conflicts when the disconnected mobile unit reconnects to the fixed server. In order to guarantee the serializability of the global transactions, we must

specify how the system will respond if those transactions generate conflicts. This thesis proposes three options to resolve conflicts.

- **Alternative Transaction**

  A disconnected mobile unit issues an alternative transaction on the top of the master transaction while it is disconnected. When conflict occurs and the system fails to commit the master transaction, the alternative transaction is submitted to the GCP. For example: a mobile user downloaded the airline ticket information before disconnecting from the network. While it is disconnected, it wants to reserve an airline ticket to Toronto from Air Canada. The local database indicates that there were only two seats left on Air Canada. Those two seats may be gone when the user reconnects to the fixed network so the system will ask the mobile user to hold an alternative ticket from Canadian Airlines. If all of the seats from Air Canada have gone during the period of disconnection, the alternative transaction will replace the master transaction.

- **Partial Re-execution**

  In a traditional optimistic concurrency control, the system aborts all conflicting transactions and re-executes them. However, we can apply compiler technology to analyze the committed mobile transactions before executing global commit. If a conflict is detected, the GCP will re-execute only the conflicting operations in the conflicting transactions. In the worst case, we re-execute all the operations of the conflicting transaction. We argue that re-executing the whole conflicting transaction in a lightly-shared system is often unnecessary and is the worst case scenario. For example: suppose that a salesman sold 100 item A and 100 item B to a customer in a remote area using the disconnected laptop computer. The salesman's laptop showed that he had enough items in stock to commit this invoice. We can treat this invoice as a transaction which consists

of two operations (*op*1: sold 100 item A; *op*2: sold 100 item B). When the salesman reconnects back to the network, the database shows that other salesmen sold some item A to another customer during the time at which the salesman was disconnected so there are not enough item A left. Thus, *op*1 generates conflict. In traditional transaction model, even though only *op*1 generates conflict, both *op*1 and *op*2 in the conflicting transaction will be re-executed. However, *op*2 does not generate any conflict and needs not be re-executed. When only two operations are involved, the re-execution costs are not onerous, but let us consider an extreme case. If the salesman sold 1000 items and only *op*1 generates conflicts, the system must re-execute 1000 operations in the transaction because of one conflicting operation. This scenario is inefficient. Using partial re-execution, only *op*1 needs to be re-executed so other operations consistently affect the database.

- **Aborting the transaction**

  In some cases, the GCP must abort the locally committed transactions, and inform the user of the abortion because of the application's constraints. For example: suppose that a mobile user wants an airline ticket to Toronto on a specific day. Before traveling, the user downloaded the airline ticket information to the laptop computer. While he is traveling, he makes a booking and the LTM committed the transaction locally because the system had shown that there were seats available. However, upon reconnecting to the fixed server, the system shows that all seats have been taken and no more seats are available in any airline company for that day. Even though the mobile user may submit two or more alternative transactions, the GCP must abort the transaction.

For the first two options, the system resolves the conflicts automatically. Users are not required to respond when the conflict is detected. Thus, the new model

supports user transparency in that the user needs not know specifics about database management to resolve conflicts. Moreover, it is not required to re-execute every conflicting operations in a transaction. Applying this new transaction model gives us a higher concurrency without violating serializability.

For the third opinion, performance depends on the application's constraints. A transaction must be aborted if it attempts to operate illegally. (For example: a user wants to use a credit card over limit.) These constraints are pre-defined in the application so it does not require special care in our transaction model.

Chapter 4

Multi-Version Timestamp Certification for Disconnection Protocol in Mobile Computing

Multi-Version Timestamp Certification for Disconnection Protocol in Mobile Computing (MVTC) is specifically designed for the disconnection protocol. The goal is to provide the highest concurrency and mobility to both disconnected mobile units and fixed servers while maintaining data consistency. MVTC consists of two phases. The 1st phase runs independently in the disconnected mobile unit, and the 2nd phase runs when the disconnected mobile unit reconnects to the fixed servers. 1st phase is called *Local Timestamp Certificate Scheme* (LTCS). LTCS is based upon [SNM85], which is a distributed certification algorithm. LTCS is responsible for local serializable transactions and guarantees their consistency properties. 2nd phase is called *Global Commit Protocol* (GCP). GCP is responsible for identifying conflicts between the disconnected mobile unit and the fixed servers, and merging the transactions running on them together. Figure 4.1 provides an overview of MVTC.

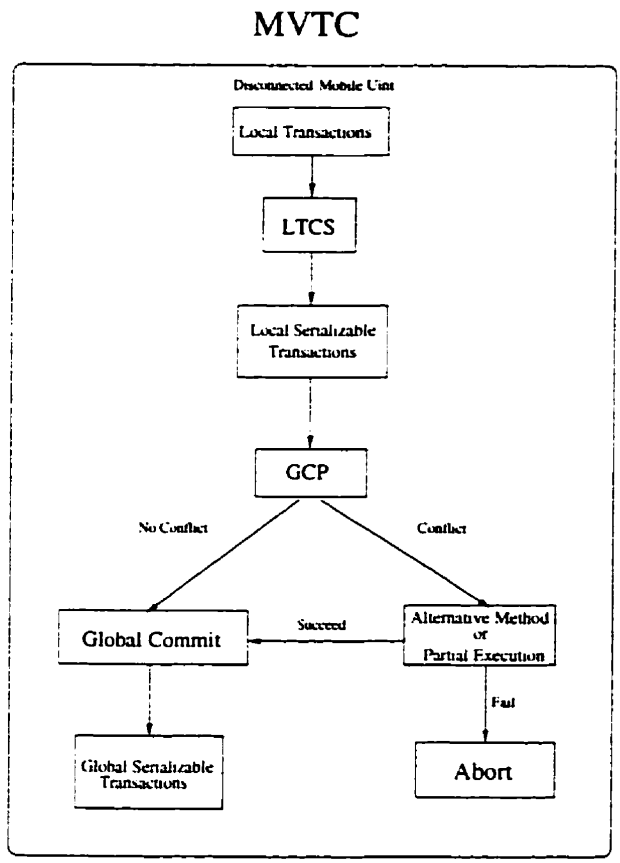


Figure 4.1: Overview of MVTC

4.1 Design Overview

MVTC is designed for an environment consisting of a large collection of fixed file servers and connected clients, and a much smaller number of disconnected moving mobile units. The design is optimized for transaction concurrency typical of lightly-shared mobile systems. It is specifically not intended for applications that exhibit highly concurrent, fine granularity, and heavily-shared system.

Each mobile unit has a local disk and is able to communicate with the fixed network over a high bandwidth network such as 10M Ethernet while it is wire connected, or over a low bandwidth network such as NCR's wavelan, Motorola's AL-

TAIR, Proxim's Range LAN and Telesystem's ARLAN [IB94] while it is moving. Mobile units may temporarily disconnect from the network and operate automatically with local data in cache. Mobile units view the fixed network as a single, location-transparent shared file system. The fixed network supports global naming and file replication. The mobile units need to know nothing about the distributed operating system underneath. Generally speaking, the mobile unit should treat the whole network as one logical file server.

4.1.1 Fixed Network

The fixed network consists of a number of file servers connected with a high bandwidth network. These file servers run 24 hours a day and reachable at any time through the network. An unreachable file server implies failure, and no valid transaction will be run on that failed machine. Compared to the mobile client, the file server is more powerful in term of computational power and storage capacity. It is relatively more durable and secure than mobile units since laptops are more fragile and vulnerable.

There are several distributed operating systems in both academic and commercial markets. Coda, Ficus and AFS [Hei92, HH93, KS92] are some examples. The ultimate goal for a distributed operating system is ensuring location transparency so clients view the whole distributed system as one large file server regardless of where they connect to the network. To achieve this goal, it involves persistent global naming, concurrency control among sites, server failure and recovery, file replication, smart caching, security, etc.

4.1.2 Mobile Client

Compared to a file server, the mobile client is less powerful in term of storage capacity and computational power. A mobile unit is equipped with a fixed disk and a com-

munication device. It can work in both connected and disconnected modes. Without network support, it is able to run transactions using the data in its cache. Users find no apparent difference in working in the disconnected mobile unit if they have previously down-loaded all the data items they need while traveling. Users require no special knowledge about how the database management system handles re-integration or how the database management system resolves and reconciles conflicting transactions.

4.2 LTCS Design Overview

The cache manager of the disconnected mobile unit acts as a pseudo-server and the cache acts as a local database. A local database executes a transaction as a process. For each process, there is a corresponding *Local Transaction Manager* (LTM). The LTM is divided into two phases. 1st phase called the *certification phase*, the LTM asks its *Certification Module* (CM) to certify each of its transactions. Several transactions are permitted to run concurrently without blocking before the commit phase. When a transaction wants to commit, its LTM will go into the *critical section* to test the serializability with respect to the other transactions running at the database. The LTM issues *certification request* to its CM and waits for a reply. After the CM certifies the transaction and responds, the LTM enters 2nd phase called the *Update Module* (UM) and installs the tentative updates of the data items permanently in its local database.

Since the LTCS guarantees only the serializability of local transactions, we must keep in mind that a set of locally certified transactions will merge to the fixed servers upon reconnection. Thus, even if a transaction gets local certification, it does not guarantee that the same transaction can commit globally. Two sets of locally certified but globally conflicting transactions may get interleaved. To resolve the global

serializability problem. the system-wide unique timestamp will be used.

4.3 LTCS Design Detail and Implementation

4.3.1 Setting a Global Clock

Since we must maintain the global serializability of transactions after reconnection. we need *system_wide* unique timestamps for comparing and merging transactions. Before disconnection. the mobile unit and the fixed server must synchronize their clocks using the "happened before" relation [Lam78] so there is no ambiguity about the time.

In distributed systems. the physical clock is not reliable because each site has its own clock which is difficult to synchronize globally. Lamport introduces a logical clock to synchronize time in distributed systems. The logical clock uses a "happened before" relation to distinguish the time. We define a process as a sequence of events. An event can be a subprogram or an execution of a system call.

The "happened before" relation denoted by "$\rightarrow$" must satisfy the following three conditions:

1. If x and y are events in the same process. and x comes before y. then $x \rightarrow y$.

2. If x is the sending of a message by one process and y is the receipt of the same message by another process. then $x \rightarrow y$.

3. If $x \rightarrow y$ and $y \rightarrow z$. then $x \rightarrow z$.

   Two distinct events x and y are said to be *concurrent* if $x \not\rightarrow y$ and $y \not\rightarrow x$.

A synchronized clock. C. is a way of assigning an order number to an event. The order number is used to indicate the sequence of which the event occurred. Lamport

defines that C_i is a function to assign an order number to an event of process i. The order number indicates the process sequence of an event. $C_i[x]$ returns an order number corresponding to an event x in process P_i. For example: events x and y belong to process P_i. If x proceeds y, then $C_i[x] < C_i[y]$. The entire system of clocks is represented by the function C. There is no direct relation of the order number to physical time. The idea of synchronized clocks in distributed systems is based upon the order in which events occur.

Lamport formally defines the *Clock Condition* as follows:

For each event x and y, we say that event x proceeds event y if

$$x \rightarrow y \text{ then } C[x] < C[y].$$

There are two *Clock Conditions* in distributed systems:

C1: For event x and y in process P_i, if x comes before y, then $C[x] < C[y]$.

C2: If event x is the sending of a message by process P_i and event y is the receipt of that message by process P_j, then $C_i[x] < C_j[y]$.

To establish C1, each process P_i increments C_i between any two successive events. To establish C2, a timestamp T_m is used on each message m. T_m equals the time at which the message was sent. Upon receiving a message timestamp T_m, a process must advance its clock to be later than T_m. For example: if event x is the sending of a message m by process P_i, then the message m contains a timestamp $T_m = C_i[x]$. Upon receiving a message m, process P_j sets C_j greater than or equal to its present value and greater than T_m.

Unfortunately, ties are still possible for two concurrent events from two different processes. To break ties, the relation "$\Rightarrow$" is used. Before defining "$\Rightarrow$", a relation "$\prec$" is defined. For any two processes P_i and P_j, $P_i \prec P_j$ means P_i has a higher

process priority than P_j. The definition of relation "$\Rightarrow$" is as follows: If x is an event in process P_i and y is an event in process P_j, then $x \Rightarrow y$ if and only if either

(i): $C_i[x] < C_j[y]$ or,

(ii): $C_i[x] = C_j[y]$ and $P_i \prec P_j$.

With the help of relation "$\Rightarrow$", we can extend the "happened before" relation to total ordering relation and synchronize the global timestamp in distributed systems. Therefore, both the disconnected mobile unit and fixed servers issue a *get_timestamp*() operation to obtain the system_wide unique timestamp. Note that transaction T_i is older than transaction T_j if $ts(T_i) < ts(T_j)$.

Global synchronization is an open problem and other researches appear in the research literature. For examples: Beguelin and Seligman [BS93] discuss the combination of the logical and physical clocks to achieve the global synchronization. Schneider [Sch86] discusses the fault-tolerant property of synchronizing global clock, and Srikanth and Toueg [ST84] present the optimal clock synchronization. Global synchronization is non-trivial to solve and how to solve it is beyond the scope of this thesis.

4.3.2 Data Model

A data item is $< name, version\_id, attrib >$ where $attrib$ is $< val, ts(r), ts(w) >$. Suppose we have a data item x, val is the current value of x: $ts(x_r)$ is the timestamp of the last transaction who reads x: $ts(x_w)$ is the timestamp of the last transaction who modifies x: $version\_id = ts(x_w)$ is used to indicate x's version. We define that data x_i is older than x_j if $verison\_id_i < version\_id_j$ $(ts_i(x_w) < ts_j(x_w))$. For each data item x, there are two queues corresponding to it: (1) *read request queue* (RR_x) and (2) *write request queue* (WR_x). These two queues are used to store all *certified requests*

(rr_i and wr_j). rr_i is created and stored in RR_x upon a successful *read certification* of transaction T_i. wr_j is created and stored in WR_x upon a successful *write certification* of transaction T_j. The structure of rr_i and wr_j is $< val.ts(r).ts(w) >$ where $rr_i.val$ ($ww_j.val$) is the value of the certified data item and $rr_i.ts(r)$ ($wr_j.ts(r)$) is the time when it is certified for *read* operation. Furthermore. $rr_i.ts(w)$ ($wr_j.ts(w)$) is the time when it is certified for *write* operation.

4.3.3 Architecture of LTCS

Figure 4.2 illustrates the architecture of LTCS.

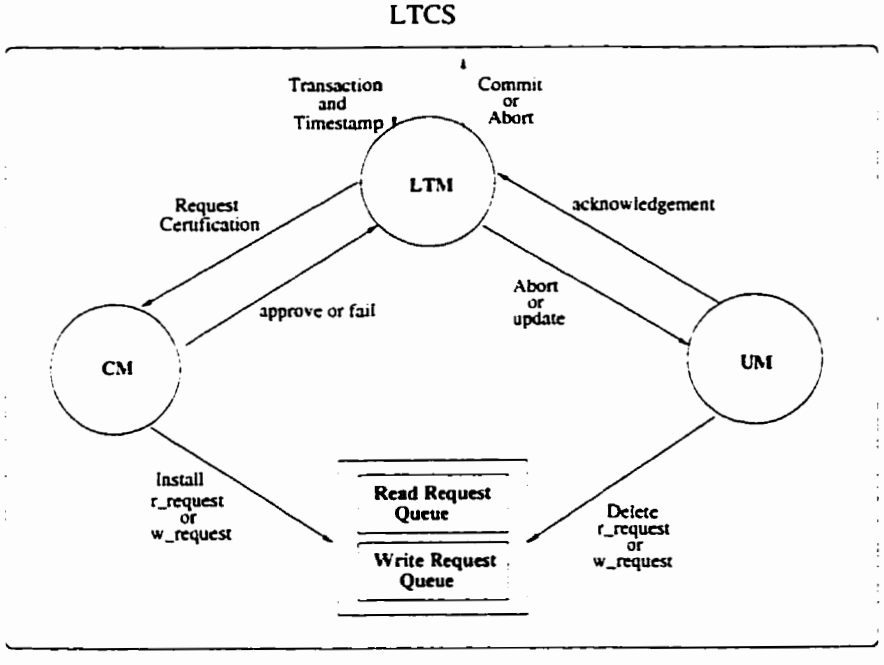


Figure 4.2: Architecture of LTCS

The LTCS is composed of three parts: (1) Local transaction manager (LTM). (2) certification module (CM). and (3) update module (UM). For each data item x, there are two queues corresponding to it (RR_x and WR_x). A transaction T_i can execute any operation before the commit time. During the execution of T_i, an *activity_table*

will be created and log all the information of all the data used in T_i. The structure of the *activity_table* is an array of records. The structure of the record entry is $< name, cur\_value, verion\_id, certify\_id, Rflag, Wflag >$. Every distinct data item x used in T_i has an entry in *activity_table* where *name* is the name of x, *cur_value* is the most recent value of x. *version_id* is the version x which has been read in *read* operation. $certified\_id$ is the timestamp when x got certified. $Rflag$ indicates x has been read, and $Wflag$ indicates x has been written. When T_i wants to commit, the LTM will first pass the *activity_table* of T_i to the CM. The CM certifies each data item x by calling *certification functions*. There are three certification functions: (1) *Certify_Read*, (2) *Certify_Write*, and (3) *Certify_RW*. *Certify_Read* is used to certify *read* operations; *Certify_Write* is used to certify *write* operations, and *Certify_RW* is used to certify *read/write* operations. A successful certification creates a *certified request*. There are two different *certified requests*, *certified read* and *certified write requests* (rr_i and wr_j). There are two queues *read request queue* (RR_x) and *write request queue* (WR_x) corresponding to every data item x, which stores all rr_i and wr_j respectively. These requests are used to detect conflict in the CM. For example: suppose that T_i gains a read certification of data x from the CM and it creates and installs rr_i in RR_x, but T_i has not yet been committed. This scenario occurs if T_i is still active. Consider another transaction T_j where T_j is older than T_i ($ts(T_i) > ts(T_j)$). T_j asks the CM to certify its write operation on x. The CM first checks RR_x and discovers that an active T_i has a read certification on x. Since T_i is younger than T_j, CM will reject T_j because T_j will overwrite the current value of x.

If CM reports a successful certification, the LTM informs the UM to update the corresponding value executed by T_i on the database permanently. Otherwise, the LTM informs the UM to abort T_i. In both cases, the UM will erase all the *certified requests* of T_i because T_i is terminated.

A data item x can be in either *normal* or *active* state. If both RR_x and WR_x are empty, x is in normal state; otherwise, it is in active state. The active state means there is at least one active transaction accessing that particular data item before commit. In the mobile client, there are separate storages for stable and unstable databases. The UM will not modify the stable database until it gains permission from its LTM.

Since the LTCS is an optimistic concurrency control mechanism, there is no locking involved in these functions except $Certify\_Read$, $Certify\_Write$ and $Certify\_RW$. At any time, the CM will permit a *Read_Data* operation. It returns the current value regardless of the state of the data item. All three certification functions are issued by the LTM to verify its transaction before commit. When the LTM issues these certification functions to the CM, it will pass its *timestamp* along with other parameters to verify its request. Next, the CM compares the input timestamp to the certified requests in the queues. If there is no conflict, the CM will send the *request_approved* to the LTM; otherwise, it sends *not_approved*.

4.3.4 Execution of Transaction

When a transaction issues **Begin_Transaction**, the LTM initializes its *activity_table* and logs all data items accessed during the execution until **End_Transaction** is issued. When the LTM submits a $Read(x)$ operation, it will check its *activity_table*. If entry x does not exist, the LTM will create an entry, and issue $read(x)$ to its CM. If x exists, it will set $Rflag$ to *true* and return the *curr_value*. Similarly, for $Write(x, new\_value)$ operation, if x does not exist in the table, the LTM will create an entry and fill it accordingly. If x exists, it will overwrite the *curr_value* with the *new_value* and set the $Wflag$ to *true*. Algorithm 4.1 and Algorithm 4.2 show the *Read* and $Write$ operations.

Algorithm 4.1 *Read Operation*

Read(x, *curr_value*)
Input x: data
Output *curr_value*: current value of x
if x exists in *activity_table* {
    set *Rflag* to **TRUE**:
    return(*cur_value*):
}
else {
        create a table entry for x:
        name = x: fetch the value of x from the database:
        fit it in *activity_table*:
        issue *read*(x):
}

End of Algorithm

The *Read* and *Write* operations of transaction T_i do not modify the actual database. They only update the *activity_table* and indicate to the LTM that a *read* or *write* operation has been performed. They leave the actual consistent database untouched until T_i commits. The CM is responsible for the certification process before T_i commits. The advantage of using the LTCS over the MVTO is the optimism. In general, *read* and *write* operations are never blocked before commit, so several transactions can run concurrently.

4.3.5 Certification Module

The following set of certification functions are issued by the LTM to the CM. Detailed implementation and explanations are given below.

Function Set 4.1 *Certification functions*

Certify_Read(*data*, *version_id*, *timestamp*)
    return (*certified* : *boolean*, *certify_id*)

Certify_Write(*data*, *new_value*, *timestamp*)
    return (*certified* : *boolean*, *certify_id*)

Algorithm 4.2 *Write Operation*

```
Write(x, new_value)
Input x: data
       new_value: new value of x

if x exists in activity_table
{
      overwrite curr_value to new_value;
      set Wflag to true;
}
else {
            create a table entry for x;
            name = x;
            set Wflag to TRUE;
            curr_value = new_value;

}
```

End of Algorithm

Certify_RW(*data*, *new_value*, *version_id*, *timestamp*)
return (*certified* : *boolean*, *certify_id*)

End of Function

When the LTM wants to certify a read operation of T_i on data x, it orders the CM to issue *Certify_Read*. *Certify_Read* first ensure that the x read before by T_i is a current version. This is done by comparing the *version_id* ($ts_a(x_w)$) of x in the *activity_table* to the *veriod_id* of x ($ts_d(x_w)$) in database. Subscript 'a' indicates *timestamp* used in the *activity_table*, and subscript 'd' indicates *timestamp* used in the database. If the current version of x in the database is younger than T_i, the CM will fetch a version of x where it contains the largest $ts_d(x_w)$ less than $ts(T_i)$ to the LTM. The only time when a certified request fails is if there exists a wr_j in the WR_x where $wr_j.ts(w) > ts(T_i)$ because a transaction T_j, where $ts(T_j) > ts(T_i)$, has already gained the write certification and waits to commit. This means that the value read by T_i earlier is stale because T_j runs concurrently with T_i and will perform a certified

write operation on x when it is committed. So the CM refuses to certify T_i's read operation. Algorithm 4.3 shows the pseudo-code of *Certify_Read*.

Algorithm 4.3 *Certify_Read*

```
Certify_Read(x, version_id, timestamp)
Input x: data to be certified
       version_id of the input data
       timestamp of calling Certify_Read
Output request_approved
       certify_id

if (ts_d(w_x) ≠ verion_id) {
     find a version of x such that ts_d(x_w) is the largest timestamp less than ts(T_i);
     update curr_value in the activity_table:
     certified =TRUE:
} else
     if (WR_x == ∅)
          certified = TRUE:
     else {
          find wr_i with oldest ts(w) from WR_x
          if (timestamp < wr_i.ts(w))
               certified = TRUE:
          else certified = FALSE:     /*due to existing certified write request*/

     }:
if (certified == TRUE)
{
     create a certified R request. rr_j where
          rr_j.ts(r_x) = timestamp:
     set x to active status:
}
```

End of Algorithm

When the LTM wants to certify a write operation of T_i on data x, it orders the CM to issue *Certify_Write*. First, it makes sure that no younger transaction performed any read operation and committed before T_i wants to certify its write operation. Second, it scans through the RR_x and finds the youngest rr_i. If $ts(T_i) < rr_i.ts(r)$, then the CM rejects T_i because a young transaction T_j has gained a certified read operation on x and not yet committed. If the CM certified T_j's write operation on x,

T_j would read stale data when it commits.

In Algorithm 4.4, there are two situations when the certified write request of T_i will be rejected.

case 1: a younger transaction T_j has read x and been committed before T_i requests the write certification on x.

case 2: a young transaction T_j has certified its *read* operation but has not committed yet. Thus, a *write* operation of T_i cannot be certified.

Algorithm 4.4 *Certify_Write*

```
Certify_Write(x, new_value, timestamp)
Input x: data to be certified
       new value of the input data
       timestamp of calling Certify_Write
Output request_approved
       certify_id

if (RR_x == ∅) {
     if (ts_d(r_x) < timestamp)
          certified = TRUE;
     else certified = FALSE;     /*case 1: due to committed transaction*/
}
else {
     find rr_i with youngest ts(r) from RR_x
     if (rr_i.ts(r) < timestamp)
          certified = TRUE;
     else cerified = FALSE;      /*case 2: due to existing certified read request.*/
}
if (certified == TRUE)
{
     create a certified W request, ww_j, where
          ww_j.val = new_value,
          ww_j.ts(r) = ts_d(r_x),
          ww_j.ts(w) = timestamp;
     set x to active status;
}
```

End of Algorithm

Algorithm 4.5 is the combination of a *Certify_Read* and a *Certify_Write* calls. On successful certification on data x, the CM creates two identical requests (rr_i and wr_i) queued in both RR_x and WR_x according to the timestamp recorded in the transaction. *Certify_RW* refuses to certify T_i if:

case 1: the transaction would overwrite a younger committed transaction.

case 2: there exists rr_j where $rr_j.ts(r)$ is older than $ts(T_i)$, or there exists wr_j where $wr_j.ts(w)$ is younger than $ts(T_i)$).

4.3.6 Update Module

The following set of functions are issued by the LTM to the UM. These two functions are used to update the database permanently. Detailed implementation and explanations will be given below.

Function Set 4.2 *Update functions*

Update_Request(data, certify_id)
return *Acknowledge*

Delete_Request(data, certify_id)
return *Acknowledge*

End of Function

After the CM verified T_i, the LTM decides to commit or abort T_i. If the CM fails to verify T_i, the LTM aborts T_i. It calls *Abort_Request* to erase all T_i's *certified requests* because T_i causes conflict. If the CM verifies T_i successfully, the LTM commits T_i. It calls *Update_Request* to update the database based on T_i's *certified requests*. Algorithm 4.6 and Algorithm 4.7 provide the pseudo-code.

Since T_i is aborted, Algorithm 4.6 erases all its *certified requests* from the RR and WR queues according to the input data name and the *certify_id*.

Algorithm 4.7 *Update_Request*

Update_Request(x.*certified_id*)
Input x: data name
Certify_id belong to x
Output Acknowledge to LTM

{
if ($rr_i \in RR_x$ and $rr_i.ts(r) = certified\_id$)
if (rr_i is not outdated) {
$ts_d(x_r) := rr_i.ts(r)$;
mark all rr_j ahead of rr_i in RR_x as outdated;
remove rr_i from RR_x;
}
else remove rr_i from RR_x;
if ($wr_j \in WR_x$ and $wr_j.ts(w) = certified\_id$)

if (wr_j is not an outdated) {
$ts_d(x_r) = max(ts_d(x_r), wr_j.ts(r))$;
$ts_d(x_w) = wr_j.ts(w)$;
mark all wr_k ahead of wr_j in WR_x as outdated;
}
else remove wr_j from WR_x;

if (no rr and wr is left in RR_x and WR_x)
state = *normal*;

Send acknowledgment to LTM;
}

End of Algorithm

4.3.7 Local Transaction Scheme

After the **End_Transaction** is issued, the LTM verifies each entry in the *activity_table*. There are two phases for the scheme: (1) Certification phase and (2) Update phase. In the certification phase, the LTM sends a *certification_request* to its CM and waits for a response. If the CM acknowledge certification, it will signal the LTM with an *certified_id*. If the CM refuses to acknowledge, it will signal the LTM to abort the transaction. In the update phase, the LTM issues a *Update_Request* to the UM upon

successful certification; otherwise, it issues *Abort_Request* to the UM. Algorithm 4.8 and Algorithm 4.9 show the detailed pseudo-codes of certification and update phases.

Algorithm 4.8 *Verification Phase*

```
Input activity_table
Output activty_table
        state: boolean to indicate successful certification

For each entry in its activity_table {
      if (Rflag and Wflag)
            send Certify_RW to CM:
      else if (Rflag)
                  send Certify_Read to CM:
            else
                  send Certify_Write to CM:

      Wait_for CM responses:
      CM will return boolean request_approved & certified_id:

      if (request_approved == FALSE)
            EXIT:
      else
            save certified_id in the activity_table:

      if (request_approved == TRUE)
      {
            signal certification_success to LTM:
            set state = certified:
      }
      else
            signal certification_failure to LTM:
            set state = abort:
}
```

End of Algorithm

Only the *verification phase* is in critical section, so the algorithm avoids certification deadlock. We have shown that LTCS maximizes optimism and guarantees synchronization and serializability. The LTCS remains functional in the mobile unit until network access is regained.

Algorithm 4.9 *Update Phase*

Input *activity_table*
Output Acknowledge to LTM

For each entry in its *activity_table* {
fetch the *certified_id* of the entry *i*:
if it is a commit signal
send *update_request*(*data*. *certified_id*)) to UM:
else
send *delete_request*(*data*. *certified_id*) to UM:
Acknowledge to the *LTM*:
End the transaction.
}

End of Algorithm

When the disconnected mobile unit reconnects to the network. the *global commit protocol* (GCP) applies. The major duty of the GCP is re-integrating the locally committed transactions in the mobile unit to the fixed servers. Upon reconnection, each committed transaction calls the GCP. Thus. we must provide a concurrency control mechanism to protect the consistency of the system. The GCP is able to synchronize the locally committed transactions and maintain the strict consistent state of the database at all time.

Algorithm 4.5 *Certify_RW*

Certify_RW(x. *new_value*. *version_id*. *timestamp*)
Input x: data wanted to be certified
new_value of the input data
version_id of the input data
timestamp of calling *Certify_RW*
Output *request_approved*
certify_id

if ($ts_d \neq version\_id$)
find a version of x such that $ts_d(x_w)$ is the largest ts less than $ts(T_i)$:
if ($WR_x == \emptyset$ and $RR_x == \emptyset$)
if ($ts_d(x_w) < timestamp$)
certified = **TRUE**:
else *certified* = **FALSE**: /*Case 1*/
else {
find wr_i with the oldest $wr_i.ts(r)$ from WR_x
find rr_j with the youngest $rr_j.ts(w)$ from WR_x
if ($timestamp > rr_j.ts(r)$ and $timestamp > wr_i.ts(w)$)
certified = **TRUE**:
else *certified* = **FALSE**: /*Case 2*/
}:

if (*certified* == **TRUE**)
{
create a rr_k and wr_k where
$rr_k.val = wr_k.val = new\_value$.
$rr_k.ts(r) = wr_k.ts(r) = timestamp$.
$wr_k.ts(w) = wr_k.ts(w) = timestamp$:
}

End of Algorithm

Algorithm 4.6 *Abort_Request*

Abort_Request(x. *certified_id*)
Input x: data name
Certify_id belong to x
Output Acknowledge to LTM

{
if ($rr_i \in RR_x$ and $rr_i.ts(r) = certified\_id$)
delete all rr_i from RR_x:
if ($wr_j \in WR_x$ and $wr_j.ts(w) = certified\_id$)
delete all wr_j from RW_x:
if (no rr **and** wr is left in RR_x **and** WR_x)
state = *normal*:
send acknowledgment to LTM:
}

End of Algorithm

Chapter 5

Global Commit Protocol

5.1 Overview of Global Commit Protocol

Upon reconnection, the GCP applies and commits the committed mobile transactions globally. The re-integration process requires: (1) Each transaction has a globally unique timestamp which eliminates ambiguity of the timestamp of transaction: (2) Each transaction has its own *activity_table* which recorded all the accessed data items used during the execution of the mobile transaction. When re-integration begins, each transaction will be submitted to the GCP serially. The GCP schedules the committed mobile transactions to commit globally. For example: suppose that transaction T_i wants to commit globally. The GCP attempts to verify T_i and detect any conflicts. If there are no conflicts between T_i and other globally committed transactions, T_i will commit. Otherwise the GCP will try to resolve the conflicts introduced by T_i. Only if the GCP resolves the conflicts successfully will T_i be committed.

5.2 Architecture of GCP

Figure 5.1 shows the main GCP components.

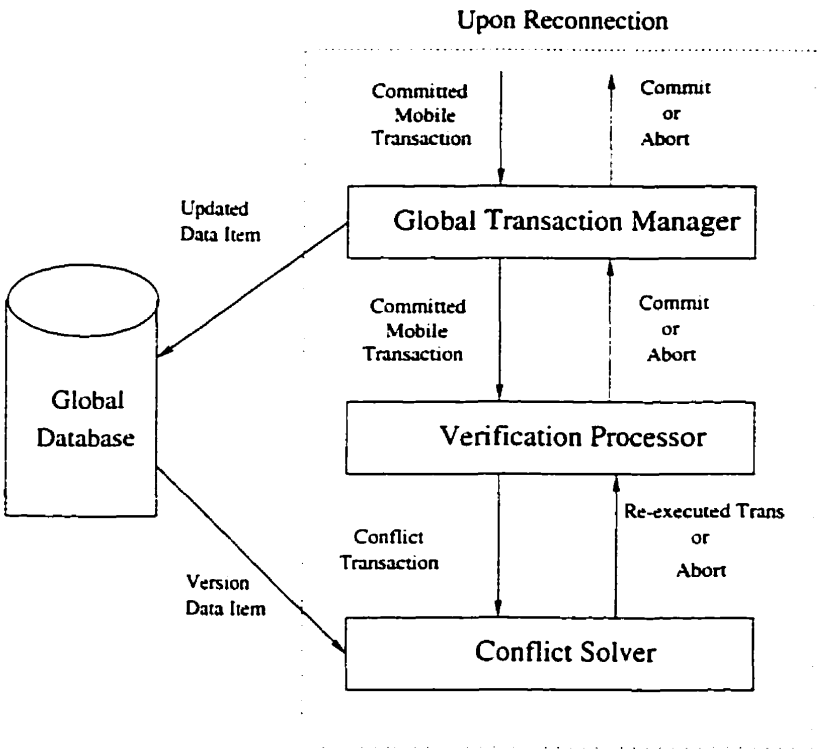


Figure 5.1: The Main Components of GCP

Three major components compose the GCP. They are: *Global Transaction Manager* (GTM), *Verification Processor* (VP) and *Conflict Solver* (CS). The GTM is called when a committed mobile transaction is submitted to the GCP. It schedules the transaction with its unique timestamp. Since the GCP is an optimistic mechanism, the GTM ensures the correctness of the transaction's execution. Therefore, the GTM calls the VP to verify the transaction. The VP first scans the *activity_table* of the transaction and compares the timestamp of each data entry to the global database. If there is no conflict, it returns to the GTM. However, if any conflicts are detected, the transaction is sent to the CS to resolve the conflict. There are three possibilities to resolve the conflict: (1) execute *alternative transaction*, (2) perform *partial re-execution* [HBG97] and, (3) *abort* the transaction. Chapter 3 discussed these three

options in our new mobile transaction model. This chapter concentrates on implementing the *partial re-execution* method. After solving conflicts, the CS returns the transaction to the VP which will commit the transaction. The VP is then responsible for making the effects of committing transactions permanent.

5.3 Partial Re-execution Module

When the VP detects a conflict, it has two options. If the mobile user has an alternative transaction, the VP will attempt to commit the alternative transaction globally. If the alternative transaction fails to commit, the VP will submit the conflicting transaction to the CS. The CS is able to re-execute only the conflicting operations in the transaction. Thus, the system is not required to re-execute every single operation in the conflicting transaction. We argue that re-executing all operations in the conflicting transaction is not necessary in a lightly-shared database system. The reason is that only a small portion of the operations in the transaction actually generates conflicts. Thus, re-executing the whole conflicting transaction is very inefficient and ineffective. The CS has been developed to analyze and identify all the conflicting operations in a conflicting transaction. Then the system re-executes only the conflicting operations to maintain the serializability. However, identifying all the conflicting operations is not a simple task. It requires the CS analyze the static dependency of each variable used in the transaction before the execution of global commit. We define that the compile time is the moment that the set of locally committed mobile transactions are being verified before the execution of global commit. For example: suppose that transaction T_i generates conflicts when it tries to commit globally. Furthermore, the VP detects that data item A is stale, the VP analyzes T_i and finds out that all the direct and indirect operations using A must be re-executed. Figure 5.2 demonstrates the effect of the partial re-execution.

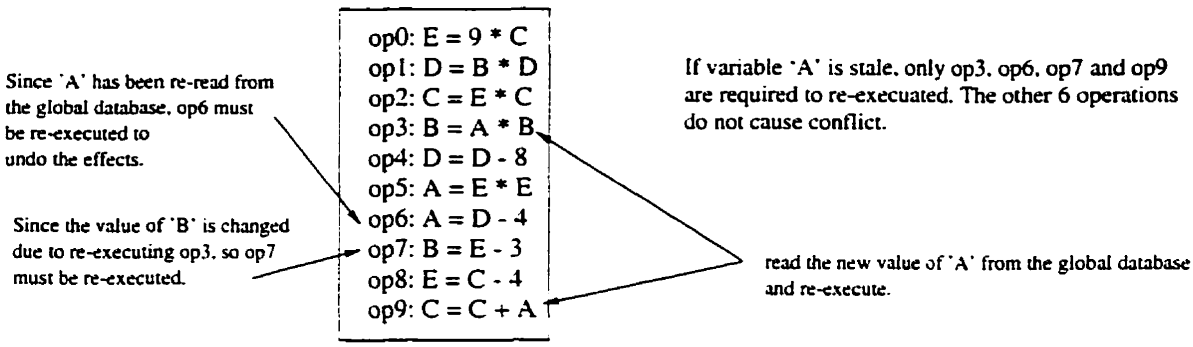


Figure 5.2: Re-executing Operations caused by GCP

The heart of the conflict solver is the mechanism of analyzing the static information of a conflicting transaction. We call it the *Partial Re-execution Algorithm.*

5.4 Data Structure for Re-execution Algorithm

Before discussing the algorithm, there are several data structures associated with the algorithms. They are as follows:

- **m**: number of *data variables* in T_i.
- **n**: number of *operations* in T_i.
- **State_Array**: is a one-dimensional array of records. Each record consists of four fields, $< result, arg1, op, arg2 >$. For example: $op_0 : E = 9 * C$, $result$ is E, $arg1$ is 9, op is $*$ and, $arg2$ is C. The size of **State_Array** is n.
- **RSet[i,j]**: is a 2D-array of integer. $RSet[i, j] = k$ means that op_i read var_j from op_k. The size of **RSet[i,j]** is $n \times m$.
- **WSet[j]**: is a one-dimensional array of integer. It records the last operation that modified var_{index}. The size of **WSet[j]** is m.

- **Re_State_Set**: is a queue to keep a set of operations for re-execution in transaction T_i.

To perform *partial re-execution*, the following set of functions is performed by the CS. Each is discussed in more detail below.

Function Set 5.1 *CS's functions*

Generate_RTable(s : *state_array*)
return (*RSet*)

Generate_WTable(s : *state_array*)
return (*WSet*)

Final_Stale_Var(x : *data*, s : *state_array*)
return (*Re_State_Set*)

Final_Related_State_in_Rset(*op* : *state*, *rs* : *Re_State_Set*
s : *state_array*, *RSet*, *WSet*)
return (*Re_State_Set*)

End of Function

Functions *Generate_RTable* and *Generate_WTable* generate *RSet* and *RWet*, respectively. Based upon the information of *RSet* and *WSet*, the GTM passes each committed mobile transaction to the VP. The VP calls functions *Final_State_Var* and *Final_Related_State_in_Rset* to identify all the necessary operations to be re-executed. The execution sequence is as follows: the VP scans and compares all the data item in the *activity_table* one by one to the same data item in the global database. If a stale data item x is found, the VP passes that stale variable to *Final_State_Var*. This function returns all the operations which directly used x in its argument and stores them in *Re_State_Set*. The VP investigates each operation in the *Re_State_Set* and calls function *Final_Related_State_in_Rset* recursively to search out all the related operations that used x indirectly and inserts them in the *Re_State_Set*. If the conflicting transaction cannot be re-executed, the VP returns *abort* to the GTM; otherwise, it

returns *commit* to the GTM which commits or aborts the transaction accordingly. Figure 5.3 illustrates $RSet$ and $WSet$ generated by functions *Generate_RTable* and *Generate_WTable* based upon the transaction shown in Figure 5.2.

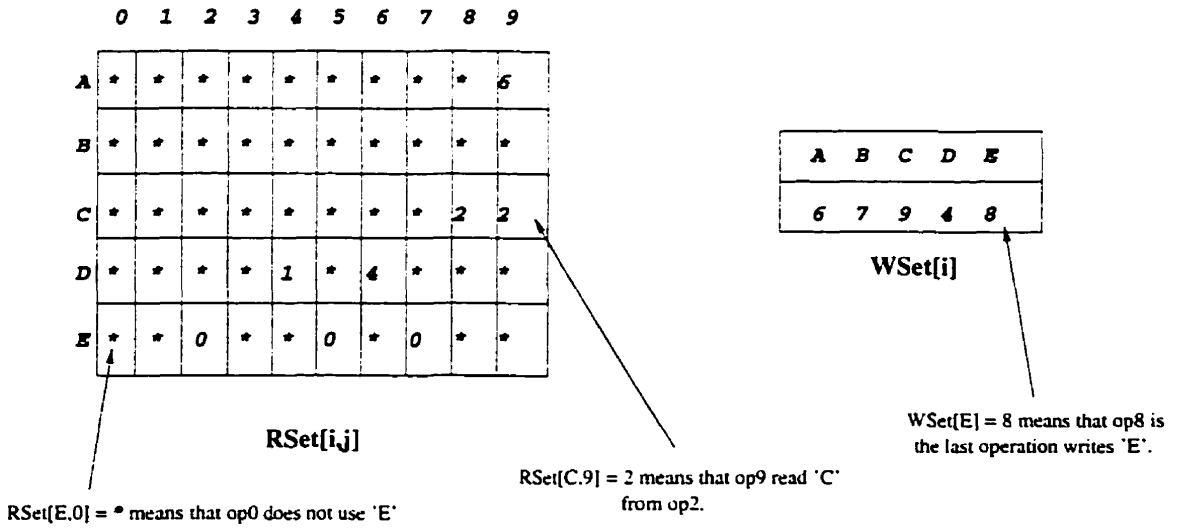


Figure 5.3: RSet and WSet based upon Figure 5.2

Entry $RSet[i, j] = k$ represents that operation op_i reads variable j from operation op_k. For example: in Figure 5.3, $RSet[9, A] = 6$ means op_9 reads 'A' from op_6. Entry $WSet[i] = k$ represents that operations op_k is the last operation that modifies variable i. In Figure 5.3, $WSet[A] = 6$ mean op_6 is the last operation which modifies 'A'.

Suppose the VP detects that T_i generates conflicts. Furthermore, the VP identifies that operation op_i read stale data and caused conflicts. There are six cases to consider when op_i is being re-executed. Suppose that transaction T_i has read stale data item B. Figure 5.4 shows these six cases.

Case 1: op_i is not related to any other operation in T_i, so only op_i is re-executed.

Case 2: op_i used B to compute A and op_s reads A. After the re-execution of op_i, A becomes stale, so op_s must be re-executed.

Case 3: op_i read stable data B so the system refreshes B from the fixed servers and re-executes op_i. However, at some later time of the transaction, op_s computes B. Since B is refreshed, the CS must re-execute op_s to compute the correct result of B. Thus, both op_i and op_s must be re-executed to guarantee correct serialization.

Case 4: op_f used B to compute A, op_i used the A computed by op_f to compute C, and op_s used the C computed by op_i to compute D. When op_f is re-executed, op_i must be re-executed because A becomes stale after re-executing op_f. When op_i is re-executed, op_s must be re-executed because C becomes stale after re-executing op_i. This case is recursive.

Case 5: When B is stale, B will be refreshed from the fixed server. However, op_i used the B produced by op_f to compute A, so the CS should re-execute op_f to undo the effect. Thus, both op_f and op_i must be re-executed.

Case 6: When op_i is re-executed, A will be overwritten, so the CS must re-execute op_s to compute the correct value of A. Thus, both op_i and op_s must be re-executed.

5.5 Detail Implementation

5.5.1 Conflict Solver

Algorithm 5.1 is used to generate array *RSet*. It scans the *State_Array* to determine the *read dependency* of the variables among the operations. Algorithm 5.1 is divided into two parts. Part I determines the read dependency of the first argument of the operation, and Part II determines the read dependency of the second argument. For example: in Figure 5.2, we want to determine the read dependency of $op_9 : C =$

$C + A$. Part I of Algorithm 5.1 determines where op_9 reads 'C' from. It scans through *State_Array* and finds out that op_2 is the last operation to modify C , so the algorithm will insert 2 in $RSet[9, C]$. Similarly, Part II of Algorithm 5.1 finds out that op_6 modified A before op_9 read it. It will insert 6 in $RSet[9, A]$.

Algorithm 5.2 generates $WSet$ from *State_Array*. It gets a variable x and starts scanning from the last operation. Once it finds $State\_Array[i].result = x$, it will stop and fill the operation number in the corresponding entry in $WSet$. For example: in Figure 5.2, we want to find out which operation is the last to modify 'A'. The algorithm scans through *State_Array* from the bottom to the top. It finds op_6 is the last to modify A, so it inserts 6 to $WSet[A]$.

Given a stale variable x, Algorithm 5.3 identifies four re-execution cases (case 1, case 3, case 5 and case 6) shown in Figure 5.4. The algorithm is divided into four parts: (1) The first **if-statement** is responsible for identifying any operation using x in its arguments directly (case 1 in Figure 5.4) and keeps it in *Re_State_Set*. (2) Once a conflicting operation is identified ($s[i]$), the algorithm will identify the last operation modifying $s[i].result$ from $WSet$ (case 6 in Figure 5.4) and keeps it in *Re_State_Set*. (3) The algorithm identifies the last operation to modify x from $WSet$ (case 3 in Figure 5.4) and keeps it in *Re_State_Set*. (4) The last **for-loop-statement** finally identifies the operation $s[i]$ read x from (case 5 in Figure 5.4) and keeps it in *Re_State_Set*.

Given the re-execution of operation op_i. Algorithm 5.4 can identify three re-execution cases (case 2, case 4 and case 6) shown in Figure 5.4. Algorithm 5.4 first scans though *RSet*. If there is an operation (op_j) that read a stale result produced by op_i, Algorithm 5.4 will call itself with input op_j recursively. The algorithm compares op_j's result ($s[j].result$) to $WSet$. If op_j is not the last operation that modified the variable (x) kept in $s[i].result$, the operation stored in $WSet[x]$ must be re-executed

(case 6 in Figure 5.4). Thus, when Algorithm 5.4 is terminated, it returns all the re-executing operations when op_i is re-executed.

5.5.2 Verification Processor

The VP is a module to identify any potential conflicts in T_i. It first checks every variable x used in T_i. If x is stale, the VP calls $Final\_Stale\_Var$ to determine all the operations that used x. The VP then calls $Final\_Related\_State\_in\_Rset$ to find all the re-executing operations in $RSet$ and $Final\_Write\_State$ to find all the re-executing operations in $WSet$. Finally, VP gets a set of re-executing operations and acknowledge the GTM if the re-execution is successful.

Algorithm 5.6 examines every re-executing operation (op_i) to ensure that re-executing op_i does not violate the application pre-defined constraints such as over-sell a product. The algorithm signals the VP '*succeed*' if op_i performs no illegal operation or '$fail$' if op_i performs an illegal operation.

5.5.3 Global Transaction Manager

Suppose transaction T_i wants to commit, the GTM calls the VP to verify T_i. If the VP returns $verify$ = **TRUE**, the GTM commits T_i; otherwise, T_i is aborted. Algorithm 5.7 shows the required pseudo-code.

5.6 GCP's Complexity

Given that m is the number of data variables in transaction T_i and n is the number of the operations in T_i, the complexity of the GCP's algorithms are as follows:

- Algorithm 5.1 is $O(n^2m)$:

  Algorithm 5.1 contains a 3-level nested **for-loop**. The outer most loop takes n

times, the second loop takes n times, and the third loop takes m times. Thus, the complexity is $O(n^2m)$. The computation is shown in Equation 5.1.

$$\begin{aligned} T(mn) &= \sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{k=1}^{m} 1 \\ &= m \cdot \sum_{i=1}^{n}\sum_{j=1}^{i} 1 \\ &= m \cdot \sum_{i=1}^{n} i \\ T(mn) &= m \cdot n(n+1)/2 \\ \therefore\ T(mn) &= O(n^2m) \end{aligned} \tag{5.1}$$

- Algorithm 5.2 is $O(mn)$:

  Algorithm 5.2 contains a 2-level nested **for-loop**. The outer loop takes m times and the inner loop takes n times. Thus, the complexity is $O(mn)$.

- Algorithm 5.3 is $O(nm)$:

  Algorithm 5.3 contains a 2-level nested **for-loop**. The outer loop takes n times and the inner loop takes m times. Thus, the complexity is $O(nm)$.

- Algorithm 5.4 is $O(m^{n-1}n^n)$:

  Algorithm 5.4 contains a 2-level nested **for-loop**. The outer loop takes m time and the inner loop takes n times. Inside the inner loop, a n-level recursion is used. Thus, the complexity is $O(m^{n-1}n^n)$. Equation 5.2 shows the recursive computation.

$$
\begin{aligned}
T(mn) &= mnT(n-1) \\
T(mn) &= mn(m(n-1))T(n-2) \\
T(mn) &= m^2n(n-1)T(n-2) \\
&\vdots \\
T(mn) &= m^{n-1} \cdot (n(n-1)\cdots 3 \cdot 2) \cdot T(n-(n-1)) \\
T(mn) &= m^{n-1} \cdot n! \cdot T(1) \\
\because \quad n^n &> n! \\
\therefore \quad T(mn) &= O(m^{n-1}n^n)
\end{aligned}
\tag{5.2}
$$

- Algorithm 5.5 is $O(m^{n+1}n^{n+2})$:

  Algorithm 5.5 contains a 2-level **for-loop**. The outer loop takes m times. $Final\_Stal\_Var$ inside the outer loop takes mn times. the inner loop takes n times. and $Final\_Related\_in\_Rset$ inside the inner loop takes $m^{n-1}n^n$ times so the complexity is:

  $$T(n) = m \cdot mn \cdot n \cdot m^{n-1}n^n = O(m^{n+1}n^{n+2})$$

- Algorithm 5.6 is $O(n)$:

  Algorithm 5.6 contains only 1 **for-loop** which takes n times. Thus. the complexity is $O(n)$.

- Algorithm 5.7 is $O(tm^{n+1}n^{n+2})$:

  Given than t is the number of transactions. Algorithm 5.7 calls Algorithm 5.5 t times. Thus. the complexity is $O(tm^{n+1}n^{n+2})$.

The overall complexity of the GCP is equal to the complexity of Algorithm 5.7 which is $O(tm^{n+1}n^{n+2})$. Thus. the complexity of the GCP is exponential.

An exponential complexity is not desirable and may not be optimal. However, the exponential complexity is only provoked by Algorithm 5.4. The main function

of Algorithm 5.4 is to find out all the dependent operations of the input operation. In a highly intra-dependent environment, Algorithm 5.4 will be called frequently and resulting in an exponential complexity. We define that the *Intra-dependency* indicates the level of data dependency of a transaction. A highly intra-dependent transaction means that the variables in the transaction are highly dependent to each other. We argue that the GCP performs well in a low intra-dependency because it reduces the chances of calling Algorithm 5.4.

In the real world, a lightly-shared and low intra-dependent environment does exist. For example, a traveling salesman traveled to a remote site to help his client. Before the salesman left the head office, he had downloaded all the necessary files in his laptop. When he finished discussing with his client, the client placed him a sales order. The order contains a list of items to be purchased. The salesman took the order and committed it in his laptop locally. Consider the data overlap problem: In the real world, it is unlikely that other salesmen in the head office would modify the same client's information concurrently. Consider the intra-dependency: When a client places an order, it is reasonable to assume that the intra-dependency of that order is low. For instance: operation 1 (op_1) sells 100 item x and operation 2 (op_2) sells 100 item y. In this case, there is no direct nor indirect relationship between op_1 and op_2. Thus, the intra-dependency is zero. Suppose that op_1 generates conflicts when the salesman reconnected his laptop to the network. Using the static analysis, the transaction manager knows that only op_1 must be re-executed and the rest of the operations consistently affect the database. Thus, the rate of calling Algorithm 5.4 to analyze operation dependency is minimal.

Using the static analysis is feasible because once the analysis is done, the transaction manager requires no overhead nor re-scheduling to solve any conflict at run time. The transaction manager can commit the transaction without worrying about any

dynamic information such as the sequence of *read/write* operations of transactions to maintain serializability.

5.7 Summary

This chapter shows that the GCP is able to synchronize the mobile transactions to the server optimistically and resolves conflicts just before commit time. Thus, this new mechanism allows the disconnected mobile unit to operate without concern for data inconsistency. Moreover, the GCP can identify and re-execute only the conflicting operations. This reduces unnecessary re-execution and thereby increasing concurrency and performance.

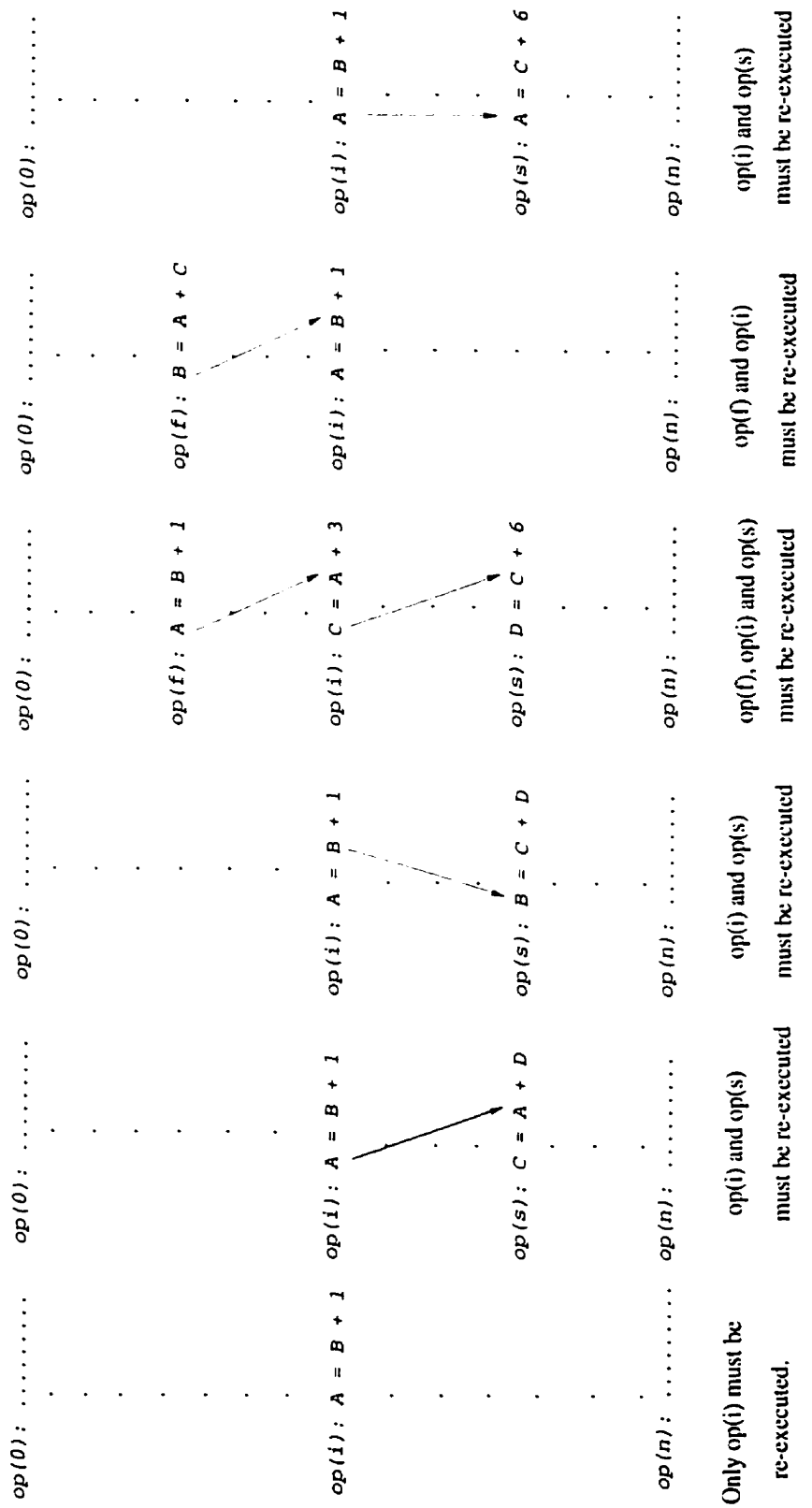


Figure 5.4: Six Cases for Partial Re-execution

Algorithm 5.1 *Generate_RTable*

```
Generate_RTable(s) : RSet:
Input s:state_array
Output RSet:Read_Set

for i = n to 1 do
      for j = i-1 to 1 do
            ch1 = s[i].arg1:
            ch2 = s[j].result:
            if (ch1 == ch2)
                  for k = 1 to m do
                        if (ch1 == k)
                              (RSet[i. k] == j):
                  endfor
      endfor

      for j = i-1 to 1 do
            ch1 = s[i].arg3:
            ch2 = s[j].result:
            if (ch1 == ch2)
                  for k = 1 to m do
                        if (ch1 == k)
                              (RSet[i. k] == j):
                  endfor
      endfor
endfor
return(RSet):
```

End of Algorithm

Algorithm 5.2 *Generate_WTable*

```
Generat_WTable(s : state_array) : WSet:
Input: state_array
Output:WSet

for i = 1 to m do
      for j = n to 1 do
            if (s[i].result == x)
                  WSet = WSet ∪ j:
      endfor
endfor
return(WSet):
```

End of Algorithm

Algorithm 5.3 *Final_Stale_Var*

Final_Stale_Var(x : *data*. s : *state_array*. *RSet*. *WSet*) : *Re_State_Set*:
Input x: stale data
state_array: all operations of a transaction
RSet: 2D array storing the read dependency
WSet: 1D array storing the final write operation
Output *Re_State_Set*: a set keeps the re-executing operations

```
for i = 1 to n do
    if (s[i].arg1 == x or s[i].arg2 == x) {
        Re_State_Set = Re_State_Set ∪ state_i:                    /*case 1*/
        Re_State_Set = Re_State_Set ∪ WSet[s[i].result]:          /*case 6*/
        Re_State_Set = Re_State_Set ∪ WSet[x]:                    /*case 3*/
        for j = 1 to m do
            if (RSet[j, i] not EMPTY)
                Re_State_Set = Re_State_Set ∪ RSet[j, i]:         /*case 5*/
        endfor
    }
endfor
return(Re_State_Set):
```

End of Algorithm

Algorithm 5.4 *Final_Related_State_in_Rset*

```
Final_Related_State_in_Rset(op : state. rs : Re_State_Set.
        s : state_array. RSet. WSet) : Re_State_Set:
Input op: re-executing operation
        rs: a set keeps the re-executing operations
        state_array: all operations of a transaction
        RSet: 2D array storing the read dependency
        WSet: 1D array storing the final write operation
Output rs: a set keeps the re-executing operations

for i = 1 to m do
    for j = 1 to n do
        if (RSet[i. j] == op) {
            if (WSet[s[j].result] != op)
                rs = rs ∪ WSet[s[j].result]:                    /*case 6*/
            rs = rs ∪ state_j:                                  /*case 2*/
            rs = rs ∪ Final_Related_State_in_Rset(RSet[i. j]. rs.
                    s. RSet. WSet):                             /*case 4*/
        }:
    endfor
endfor
return(rs)
```

End of Algorithm

Algorithm 5.5 *Verification_Processor*

Verification_Processor(*activity_table* of T_i)
Input *activity_table* of T_i
Output *verify*: a boolean flag to indicate successful verification

```
for each entry x in activity_table of T_i do
    if (Rflag == TRUE) {
        if (version_id(x) < version_id_global(x)) {
            if (alternative transaction fails to commit) {
                Re_State_Set = Final_Stale_Var(x.state_array);
                for every op_i ∈ Re_State_Set do
                    Re_State_Set = Final_Related_State_in_Rset(op_i,
                        Re_State_Set, RSet);
                endfor
            };
            else
                submit the alternative transaction;
        };
    };

    if (Re_State_Set ≠ ∅)
        Re_execute(Re_State_Set.done);
        if (done == succeed)
            return(verify = TRUE);
        else
            return(verify = FALSE);
    else
        return(verify = TRUE);          /*no re-execution is required*/
endfor
```

End of Algorithm

Algorithm 5.6 *Re_execute*

Re_execute(*Re_State_set*. *succeeded*):
Input *Re_State_Set*: a set of re-executing operations
Output *done*: a boolean to indicate a successful re-execution

for every element $op \in Re\_State\_Set$ **do**
    if (op_i is not illegal)
        execute(op_i):
    else {
        return($done = fail$):
    }
end for
return($done = succeed$):

End of Algorithm

Algorithm 5.7 *Global Transaction Management*

$Global\_Transaction\_Manager(T_i)$
Input Transaction T_i

for every T_i enters to the GCP **do**
    Call $Verification\_Processor(activity\_table$ of $T_i.verify)$:
    if ($verify ==$ **TRUE**)
        $global\_commit(T_i)$
    else
        $abort(T_i)$:
endfor

End of Algorithm

Chapter 6

Demonstration in Mobile Computing

The GCP has been simulated using C++. The model is able to generate a random number of operations and variables. The variables are limited to 52 letters ('a' - 'z' and 'A' - 'Z'), so a transaction can access at most 52 variables. Moreover, we can input the percentage of the data overlap between mobile and fixed servers. The simulation can generate transactions and identify which operations in a mobile transaction are required to be re-executed. From the simulation, we measure the GCP's performance in two aspects: (1) Re-execution Rate, and (2) Time Complexity.

6.1 Re-execution Rate

When we analyze the re-execution rate of the GCP, we conclude that there are two factors affecting the performance: (i) percentage of the data overlap between mobile and fixed servers, and (ii) intra-dependency of the operations in a transaction.

6.1.1 Data Overlap

The GCP is an optimistic concurrency control scheme, so it will not work well in a heavily-shared system. If the data items are shared heavily between the fixed servers and the disconnected mobile unit, we can expect a lot of conflicts. In our experiments, we set up three different sets of data. Each set of data represents a different number of operations in each transaction: Set 1 contains at most 10 operations, Set 2 contains at most 20 operations, and Set 3 contains at most 50 operations. Table 6.1 shows the results. In Table 6.1, *Overlap%* is the percentage of data overlap between the disconnected mobile unit's and fixed servers' transactions; OP is the average number of operations generated for 10 transactions; RE is the average number of operations re-executed for 10 transactions, and % is the overall percentage of re-execution.

GCP's Performance for 10 Transactions									
Overlap%	Set 1 (10)			Set 2 (20)			Set 3 (50)		
	OP	RE	%	OP	RE	%	OP	RE	%
10	8.0	1.4	17.50	14.7	2.5	17.01	35.3	7.3	20.68
20	6.9	1.3	18.84	16.4	3.4	20.73	41.8	16.4	39.23
30	7.5	2.0	26.67	17.9	4.6	25.69	37.2	16.4	44.09
40	8.2	3.5	42.68	15.7	8.5	54.14	42.9	33.8	67.88
50	7.8	3.8	48.72	13.9	8.8	63.10	33.0	22.4	78.79
60	8.0	4.5	56.25	15.3	10.6	68.94	34.1	29.4	86.22
70	8.7	6.8	78.16	14.3	10.5	69.28	38.7	34.7	89.66
80	8.0	7.2	90.00	13.2	9.1	73.43	36.3	33.8	93.11
90	8.4	7.6	90.48	13.6	12.6	92.65	32.8	31.5	96.04

Table 6.1: GCP's Result for 10 Transactions

Figure 6.1 plots the results of Table 6.1. As we can see from Figure 6.1, when the

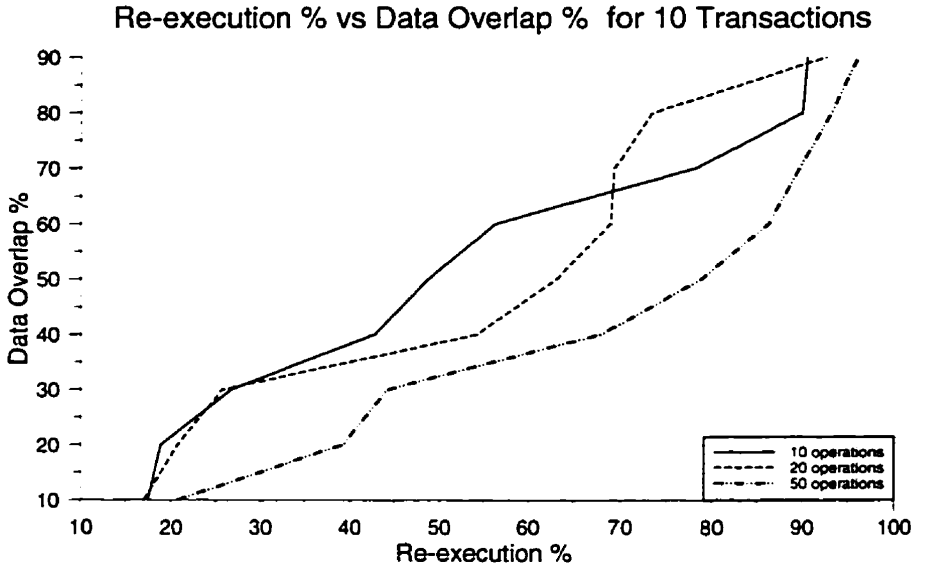


Figure 6.1: Running 10 Transactions

overlap rate is 10%, about 20% of the operations are re-executed. However, when the overlap rate increases to 50%, more than half of the operations are re-executed. Note also that when the overlap rate is set to 100%, not every operation is required to be re-executed. For example: a mobile transaction T_i contains $op_r : A = 100 + 200$ and T_i does not contain any operation reading A. Although A is stale, op_r is not required to be re-executed. When we increase the number of transactions from 10 to 100, we generate very similar results. Results are shown in Table 6.2. In Table 6.2, Set 1 contains at most 10 operations per transaction, Set 2 contains at most 20 operations, and Set 3 contains at most 50 operations. Figure 6.2 plots the results of Table 6.2.

6.1.2 Intra-Dependency of Transaction

The intra-dependency represents the operation dependency inside a transaction. A highly intra-dependent transaction means that the variables in the transaction are

GCP's Performance for 100 Transactions									
Overlap%	Set 1 (10)			Set 2 (20)			Set 3 (50)		
	OP	RE	%	OP	RE	%	OP	RE	%
10	8.06	1.55	19.23	15.61	2.45	15.70	37.22	10.16	27.30
20	7.79	2.16	27.73	15.46	5.71	36.93	39.02	14.44	37.01
30	8.30	3.20	38.55	23.05	10.44	45.29	39.24	20.16	51.38
40	7.92	3.98	50.25	16.38	9.61	58.67	38.88	26.40	67.90
50	8.37	5.09	60.81	16.17	10.17	62.89	36.80	29.60	80.43
60	7.86	5.66	72.01	15.49	10.98	70.88	36.88	30.38	82.38
70	7.89	5.94	75.29	15.62	12.74	81.56	36.73	31.59	86.01
80	8.10	7.02	86.67	15.03	12.63	84.03	37.00	34.64	93.62
90	8.13	7.26	89.30	14.97	13.46	89.91	37.36	35.74	95.66

Table 6.2: GCP's Result for 100 Transactions

highly dependent to each other. The intra-dependency plays a critical rule in the re-execution rate. If the variables used in a transaction are highly related, we can expect the rate of re-execution to be high because a re-execution of one operation will result an exhaustive search for all other affected operations. This search will go recursively and take exponential time to compute. We have shown the complexity in Section 5.6.

For example: we generate 5 transactions with at most 2,000 operations in each transaction and limit the number of variables to 10. Therefore, the operations are closely related to each other. The result is a very high re-execution rate even in a low data overlap environment. Thus, we conclude that the percentage of data overlap alone cannot determine the re-execution rate of the GCP. The GCP does not perform better in a low percentage of data overlap but closely related transaction. In contrast,

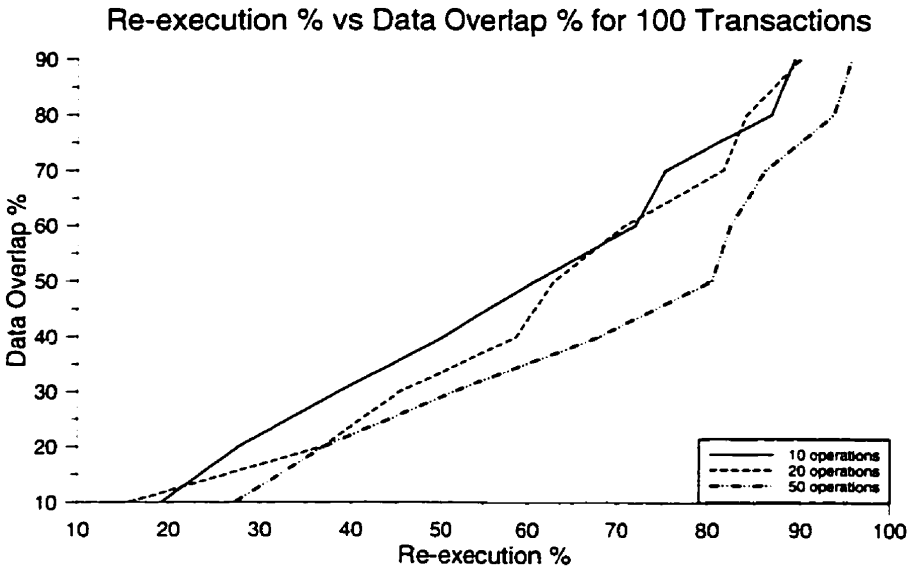


Figure 6.2: Running 100 Transactions

the level of intra-dependency between operations affects the rate of re-execution. Table 6.3 shows the result of the simulation.

Figure 6.3 plots the results of Table 6.3. Even in a low overlap scenario (10%) the re-execution rate is very high (90.28%) (see Figure 6.3). This result shows that a transaction with high intra-dependency will result a high re-execution rate. So, the GCP does not work well in highly intra-dependent transactions. In contrast, if a transaction is low in intra-dependency, the re-execution rate will be lower. Consider an extreme case shown in Figure 6.4. In Figure 6.4, T_i has zero intra-dependency because no variable in T_i is related. If we set 10% of data to stale (2 variables are stale), the maximum number of re-executing operations is two because the worst case will be two different operations read these two stale variables in their arguments. Except for these two operations, no other operations will be affected.

GCP's Performance for 2000 Operations			
Overlap%	OP	RE	Re-execution%
10	1750	1580	90.28
20	1698	1552	91.40
30	1009	934	92.57
40	1265	1200	94.86
50	1726	1642	95.13
60	1232	1195	97.00
70	1875	1819	97.01
80	1994	1950	97.80
90	1683	1678	99.70

Table 6.3: GCP's Result for 2000 Operations

6.2 Time Complexity

Similarly, we conclude that there are also two factors affecting the time complexity: (i) size of the mobile transaction, and (ii) intra-dependency of a transaction.

6.2.1 Size of Transaction

The longer the transaction, the more time the GCP takes to compute the result. However, the size of a transaction only plays a minor factor to the time performance of the GCP. We have shown in section 5.6, the complexity of the GCP is exponential. This exponential growth is contributed by the intra-dependency but not by the transaction's length. If we fix the data overlap percentage to a constant, a long transaction with low intra-dependency will run faster than a long transaction with high intra-dependency. Thus, we concentrate on analyzing the *Intra-dependency*.

Figure 6.3: Running 2000 Operations in Each Transaction

6.2.2 Intra-Dependency

When we measure the time complexity of the GCP, the intra-dependency must be considered because it dominates the running time in the long run of the GCP. When a transaction with little intra-dependency exists among its operations, the GCP can compute the result rapidly. In contrast, if a transaction has a high intra-dependency among its operations, the GCP must spend a lot of time finding all the direct and indirect operations for re-execution. In a highly intra-dependent environment, the task will result in many recursive calls of Algorithm 5.4 that lowers the GCP's performance.

In Table 6.4, we simulate the GCP with different numbers of operations. Figure 6.5 plots the result of Table 6.4. From the experiment's results, the GCP's performance drops significantly when the number of operations in a transaction is over 7000. Since our simulation only allows at most 52 variables in a transaction, a long transaction

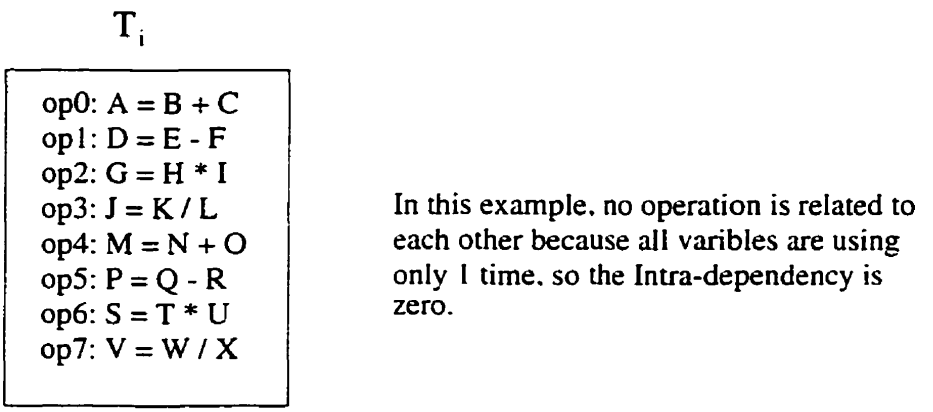


Figure 6.4: A Transaction with Zero Intra-Dependency

will result very high intra-dependency. Thus, the GCP requires much more time to compute the results.

6.3 Summary

There are two aspects to measure the performance of the GCP: (1) Re-execution Rate, and (2) Time Complexity. In a lightly-shared system with short transactions, our simulation shows that the GCP performs very well. We argue that the cut-off point for using the GCP approach is 30% of data overlap. Moreover, our simulation shows that the "*Intra-dependency*" among a transaction is an important factor to be considered. A transaction with high intra-dependency results in a high rate of re-execution and requires a long time to finish.

Size of Transaction	Time
1000	31 sec
2000	102 sec
3000	217 sec
4000	360 sec
5000	766 sec
6000	1603 sec
7000	3420 sec
8000	6012 sec
9000	12561 sec
10000	50569 sec

Table 6.4: GCP's Execution Time for Different Number of Operations

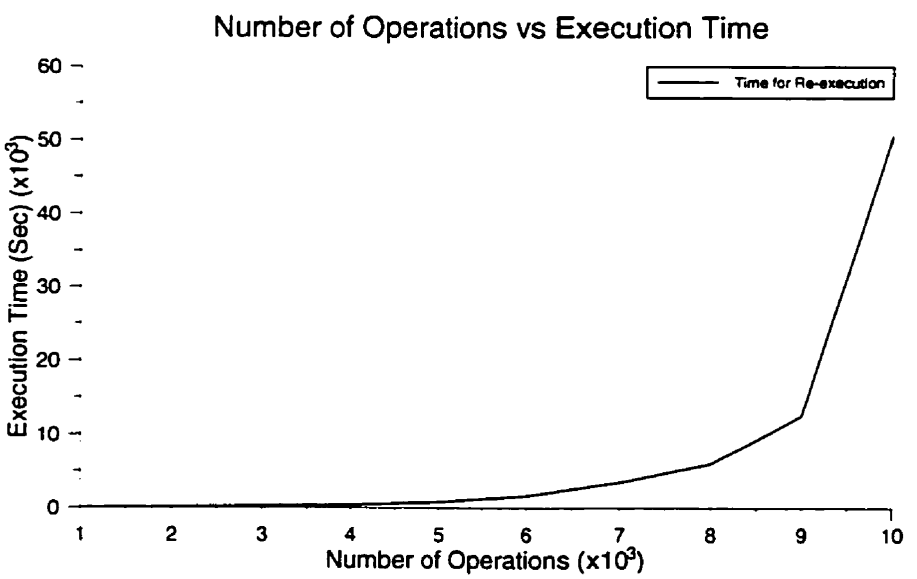


Figure 6.5: The Graph for Number of Operations vs Number of Operations

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This thesis has presented a new method for concurrency control for the disconnection protocol in mobile computing. The main property of the proposed method is to provide the highest concurrency, mobility and user transparency, as opposed to methods which limit the concurrency level and require human interaction to resolve conflicts. This thesis is divided into five parts:

1. Chapter 1 identifies the properties of mobile computing environment and presents the problem domain of allowing concurrent data access between the fixed servers and disconnected mobile unit.

2. Chapter 2 reviews the research literature and discusses briefly about their work and limitations which motivate this thesis.

3. Chapter 3 revises the traditional distributed transaction model and presents a new mobile transaction model for the disconnection protocol.

4. Chapter 4 and 5 present the detailed design and implementation of the MVTC.

5. Chapter 6 shows the actual results of the new model. From the results, we analyze and show its performance and limitations.

Performance of the GCP has been tested and presented. From the results, we argue that MVTC is feasible in a lightly-shared system. There are two advantages of MVTC over other methods: (1) user transparency, and (2) partial re-execution.

7.1.1 User Transparency

All the existing methods require some human interaction to resolve conflicts. In contrast, MVTC is fully automatic. The only time it fails to commit a transaction is when a user's transaction is illegal based on application's pre-defined constraints. For example: an attempt to withdraw money exceeding an account's limit. If conflict occurs in a transaction, the MVTC can identify it locally and globally, subsequently resolves the conflicts automatically. Users are not required to have special knowledge to use the MVTC, so it provides user transparency. For example: the travel agent in Chapter 1 needs not know how to solve the air-ticket's problem. If there is no more space in United Airlines, the transaction manager will submit the alternative transaction (to book a ticket from American Airlines) and commit the transaction. The transaction manager handles the conflict by itself alone.

7.1.2 Partial Re-execution

When concurrent data access between the fixed servers and disconnected mobile unit is permitted, past work has resolved conflicts by re-executing the whole conflicting transaction. This thesis has shown that this is sub-optimal. We have presented the GCP to detect and partially re-execute only the conflicting operations in a conflicting transaction. This thesis shows how to apply compiler technology to analyze the static information and reconcile committed mobile transactions to the fixed servers. In our

experiments, only half of the operations are required to be re-executed when a system shares 30% of its data with both the fixed servers and disconnected mobile unit during re-integration.

7.2 Future Work

There are five areas not covered in this thesis. (1)Reliability. (2) Failure detection. (3) Formal Measurement of Intra-dependency. (4) the GCP for peer-to-peer service. and (5) the GCP for multiple disconnected mobile units.

7.2.1 Reliability

Our model does not discuss any issue related to the reliability of the system. A recovery technique should be developed to ensure reliability. Recovery from transaction failure usually means that the database is restored to a consistent state after a failure. An obvious way to achieve this is to roll back and redo all the uncommitted transactions during failure. However, it is very costly, so a recovery algorithm can be developed which logs the transaction information in the GCP to minimize cascading roll back during failure.

7.2.2 Failure Detection

This thesis assumes that an unreachable mobile unit is working in the disconnected mode. However, if a mobile unit is actually failed, the system should be able to determine it. The system alone cannot tell whether a mobile unit is working in the disconnected mode or fails to operate because the system cannot reach the unit. The mobile unit should inform the fixed system when it expects to travel prior to disconnection. If the mobile unit is unable to reconnect back to the fixed servers on

schedule. the system will consider it as failure.

7.2.3 Formal Measurement of Intra-dependency

In the simulation of the GCP. we identify that the intra-dependency is one of the factors to determine the GCP's performance. The result shows that increasing the intra-dependency will lower the GCP's performance. but we do not explicitly and formally state the relationship between the intra-dependency and the GCP's performance. A formal measurement can be developed to indicate the level of intra-dependency affecting the GCP's performance.

7.2.4 The GCP for Peer-to-Peer Service

In our model. we do not allow two disconnected mobile units to exchange their information. However. this requirement is too restrictive when two disconnected mobile units wants to share some critical data in an isolated area. A modified version of the GCP (*GCP for Peer-to-Peer service* (GCP-PP)) can be developed to allow two disconnected mobiles to share their data without the fixed servers' present. Upon reconnection. the GCP-PP should be able to reconcile the transactions. Figure 7.1 illustrates the architecture of the GCP-PP.

In Figure 7.1. a disconnected mobile unit MU_1 shares data with another disconnected mobile unit MU_2. Upon reconnection. the GTM of the GCP-PP should be able to identify those shared data and reconcile the committed mobile transactions globally.

7.2.5 The GCP for Multiple Disconnected Mobile Units

In our model. we only address the problem of reconcile one disconnected mobile unit to the fixed servers at a time. However, two or more disconnected mobile units may

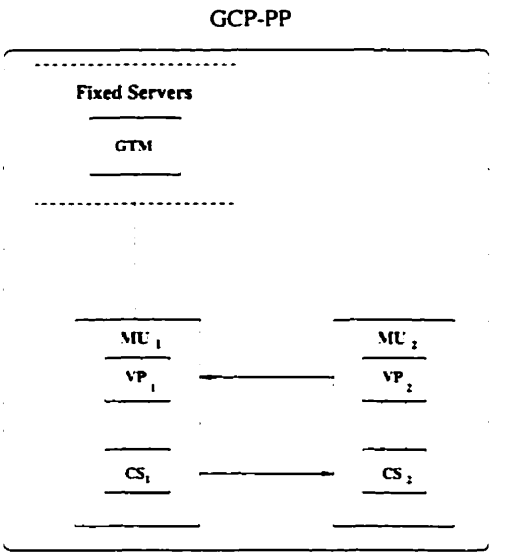


Figure 7.1: Architecture of GCP-PP

want to reconnect back to the fixed servers at the same time. In our model, we only allow them to re-integrate back to the fixed servers serially. Thus, no concurrency is involved. So, a modified version of the GCP, *GCP for Multiple Disconnected Mobile Unit* (GCP-MU), can be developed to handle the concurrency control of multiple re-integration. The architecture of the GCP-MU is shown in Figure 7.2.

In Figure 7.2, the GTM of the GCP-MU accepts and reconciles several transactions from different disconnected mobile units at the same time. These committed mobile transactions might access the same set of data, so conflicts may occur among them. The GCP-MU is responsible for detecting conflicts and reconcile those transactions concurrently. Moreover, it should guarantee the serializability among them. Maintaining serializability involves transaction scheduling and conflict detection among those disconnected mobile transactions. In addition, the GCP-MU should be able to detect and avoid deadlock.

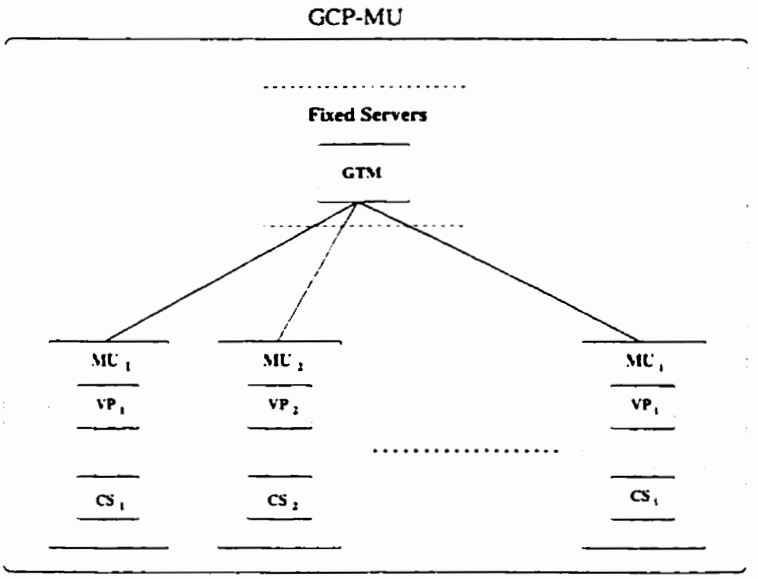


Figure 7.2: Architecture of GCP-MU

Bibliography

[BAI93] B. Badrinath. A. Acharya. and T. Imielinksi. Impact of Mobility on Distributed Computations. *Operating System Review.* 2(27):15–20. April 1993.

[BG80] P. Bernstein and N. Goodman. Timestamp based Algorithms for Concurrency in Distributed Database Systems. In *Proceedings 6th International Very Large Database.* pages 285–300. October 1980.

[BS93] A. Beguelin and E. Seligman. Causality-Preserving Timestamps in Distributed Programs. Technical Report CMU-CS-93-167. Carnegie Mellon University. June 1993.

[Chr93] P. Chrysanthis. Transaction Processing in Mobile Computing Environment. In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed System.* pages 77–83. October 1993.

[Duc92] D. Duchamp. Issues in Wireless Mobile Computing. In *IEEE International Proceeding 3rd Workshop on Workstation Operating System.* pages 1–7. Key Biscayne. FL. April 1992.

[EJB95] A. Elmagarmid. J. Jing. and O Bukhres. An Efficient and Reliable Reservation Algorithm for Mobile Transactions. In *Proceedings of 4th Inter-*

national Conference on Inforamtion and Knowledge Management, pages 90–95. Vancouver. B.C.. 1995.

[GB95] P. Graham and K. Barker. Improved Scheduling in Object Bases Using Statically Derived Information. *The International Journal of Microcomputer Applications*. 14(3):114–122. 1995.

[GBH96] P. Graham. K. Barker. and A. Hadaegh. Disconnected Objects: Reconciliation in a Nested Object Transaction Environment. In *ECOOP'96 Workshop on Mobility and Replication (WMR)*. Linz. Austria. July 1996.

[HBG97] A. Hadaegh. K. Barker. and P. Graham. Partial Re-Execution: Complex Reconciliation of Transactions to increase Concurrency in Objectbases. In *(submitted to) Principles of Database Systems*. May 1997.

[Hei92] J. Heidemann. Primarily Disconnected Operation: Experiences with Ficus. In *Proceedings of the 2nd IEEE Workshop on the Management of Replicated Data*. pages 2–5. Monterey. November 1992.

[HH93] L. Huston and P. Honeyman. Disconnected Operation for AFS. In *Proceedings of the 1993 USENIX Symposium on Mobile and Location-Independent Computing*. pages 1–10. Cambridge. MA. August 1993.

[HH94] D. Huizinga and K. Heflinger. Experience with Connected and Disconnected Operations of Portable Notebook Computer Systems. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*. pages 119–123. Santa Cruz. December 1994.

[HH95] P. Honeyman and L. Huston. Communications and Consistency in Mobile File Systems. In *IEEE Personal Communications, Special Issue on Mobile Computing*. December 1995.

[IB93] T. Imielinski and B. Badrinath. Data Management for Mobile Computing. *SIGMOD Record*. 1(22):34–39. March 1993.

[IB94] T. Imielinksi and B. Badrinath. Mobile Wireless Computing: Challenges in Data Management. *Communication of ACM*. 37(10). 1994.

[JBE95] J. Jing. O. Bukhres. and A. Elmagarmid. Distributed Lock Management for Mobile Transactions. In *Proceedings of the 15th International Conference on Distributed Computing System*. pages 118–125. Vancouver. BC. June 1995.

[KR81] H. Kung and J Robinson. On Optimistic Methods for Concurrency Control. *ACM Transaction Database Systems*. 2(6). June 1981.

[KS92] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer System*. 10(1):3–25. February 1992.

[Lam78] L. Lamport. Time. Clock. and Ordering of Event in a Distributed System. *Communications of the ACM*. 21(7):558–565. July 1978.

[LS94] Q. Lu and M. Satyanarayanan. Isolation-Only Transactions for Mobile Computing. *Operating System Review*. pages 81–87. April 1994.

[OV90] M. Özsu and P. Valduriez. *Principles of Distributed Database System*. Prentice Hall. 1990.

[Pap84] Papadimitriou. On Concurrency Control by Multiple Versions. *ACM Transactions on Database Systems*. 9(1):89–99, March 1984.

[PB93] E. Pitoura and B. Bhargava. Dealing with Mobility: Issues and Research Challenges. Technical Report CSD-TR-93-070. Purdue University. November 1993.

[PB94a] E. Pitoura and B. Bhargava. Building Information System for Mobile Environment. In *Proceeding of the International Conference on Information and Knowledge*. pages 371–378. Gaithesburg, MD. November 1994.

[PB94b] E. Pitoura and B. Bhargava. Revising Transaction Concepts for Mobile Computing. In *Proceedings of the IEEE Workshop on Mobile Systems and Applications*. pages 164–168. San Cruz. CA. December 1994.

[PB95] E. Pitoura and B. Bharagava. Maintaining Consistency of Data in Mobile Distributed Environment. In *Proceedings of 15th International Conference on Distributing Computing System*. pages 404–413. Vancouver. BC. May 1995.

[Sch86] F. Schneider. A Paradigm for Reliable Clock Synchronization. Technical Report TR86-735. Cornell University. February 1986.

[SKM+93] M. Satyanarayanan. J. Kistler. L. Mummert. M. Ebling. P . Kumar. and Q. Lu. Experience with Disconnected Operation in a Mobile Computing Environment. In *Proceedings of the 1993 USENIX Symposium on Mobile and Location Independent Computing*. pages 11–28. Cambridge. MA. August 1993.

[SNM85] M. Sinha. P. Nanadikar. and S. Mehndirtta. Timestamp based Certification Schemes for Transaction in Distributed Database Systems. In *Proceedings ACM SIGMOD International Conference on Management of Data*. pages 402–411. May 1985.

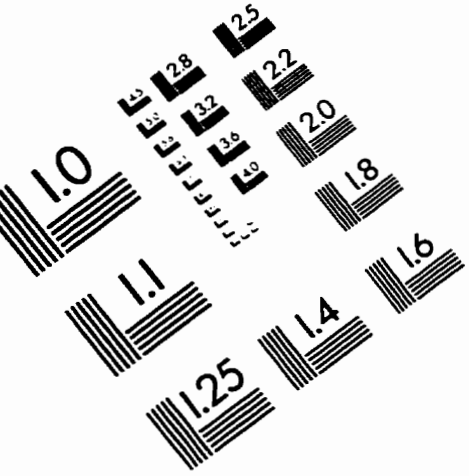
[ST84] T. Srikanth and S. Toueg. Optimal clock synchronization. Technical Report TR-84-656. Cornell University. December 1984.

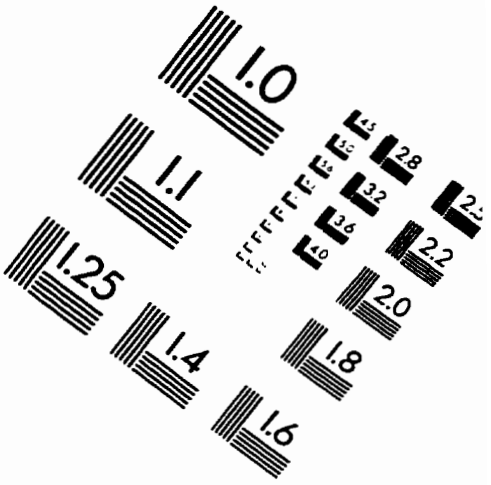
[TD91] C. Tait and D. Duchamp. Service Interface and Replica Management Algorithm for Mobile File System Clients. In *Proceedings of the 1st International Conference on Parallel and Distributed Information System.* pages 190–197. 1991.

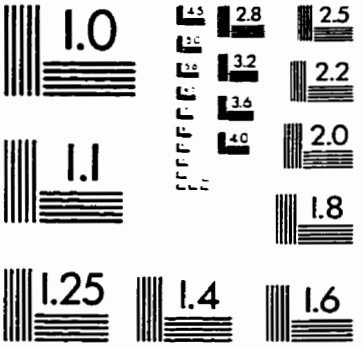
[Tho78] R. Thomas. A Solution to the Concurrency Control Problem for Multiple Copy Databases. In *Proceedings COMPCON Conference.* pages 56–62. N.Y., 1978.

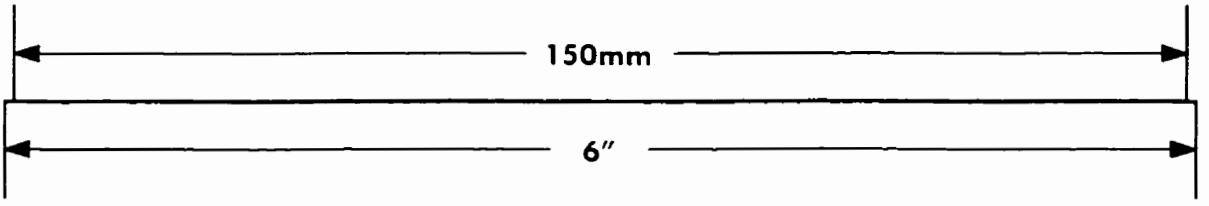
[ZD93] E. Zadok and D. Duchamp. Discovery and Hot Replacement of Replicated Read-Only File Systems. with Application to Mobile Computing. In *Proceedings of the 1993 Summer USENIX.* pages 69–85. Cincinnati. June 1993.

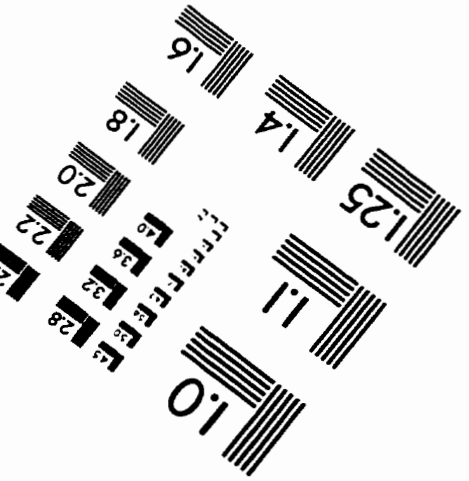
IMAGE EVALUATION
TEST TARGET (QA-3)

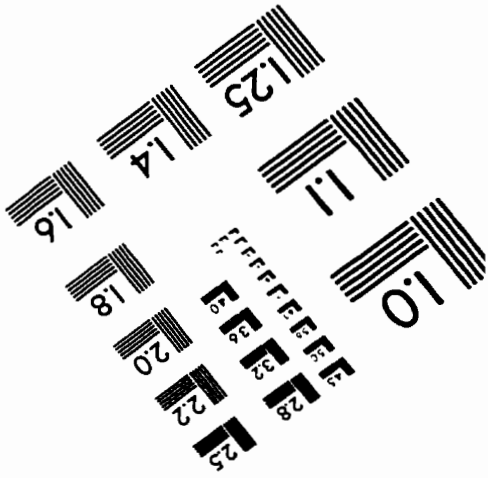












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989