A TRANSACTION EXECUTION MODEL FOR MOBILE COMPUTING ENVIRONMENTS

KALEEM A. MOMIN

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

A Transaction Execution Model for Mobile Computing Environments

by

©Kaleem A. Momin

A thesis submitted to the School of Graduate Studies

in partial fulfillment of the requirements

for the degree of

Master of Science

Department of Computer Science

Memorial University of Newfoundland

August 1999

St. John's Newfoundland Canada

To my dear little sister, Naseema Momin.
You will always be in our hearts.

Abstract

A mobile computing environment is characterized by limited execution capability at the mobile hosts, low bandwidth and the relatively high costs of wireless connection, and frequent disconnections and mobility of the mobile hosts. Such an environment naturally suggests an optimistic mode of execution, where the mobile host caches data and does the computation in disconnected mode and, on reconnection, the transaction is either committed or aborted based on the current values in the fixed network.

We propose a new transaction execution model, based on optimistic concurrency control mechanism, which dynamically adjusts the transaction execution status at the mobile host to be consistent with the database state on the mobile support station. This increases the possibility of the transaction to commit successfully and hence makes the computation on the mobile host more meaningful. A detailed algorithm is presented and its adaptability to various aspects of the mobile environment discussed.

We further strengthen the computation at the mobile host by facilitating partial guarantee against invalidation. This is accomplished by using a flexible concurrency control scheme which integrates optimistic and pessimistic approaches to access data items based on Read/Write and Write/Write-conflicts.

Keywords : Mobile computing, Transaction processing, Concurrency control, Optimistic approach, Re-execution.

Acknowledgements

Thanks to the Almighty Allah for everything.

I would like to thank my advisor, Dr. Krishnamurthy Vidyasankar, for his continuous guidance and suggestions. I could not have had a better mentor. He always made time to see me, no matter how busy his schedule. He taught me the importance of critical thinking and validating one's ideas. He has always been a constant source of inspiration.

I would also like to thank the examiners Dr. Panos K. Chrysanthis and Dr. Sanjay Kumar Madria whose valuable comments helped to improve the thesis presentation.

I would like to thank my parents Razack Momin and Ameena Bee for their moral support, my sister Naz Parveen, and my brothers Dr. Jaleel Momin, Kalam Momin and Samad Momin who not only inspired me but spent hours together on long distance phone calls making me to feel at home in St. John's. Thanks guys, I could not have better brothers and sister.

Many thanks to the faculty and staff of Computer Science department for their assistance and support throughout my program at MUN.

I would also like to thank my fellow graduate students Yunqing Gu, Ning Zhong, Vasantha Adluri, Ziqing Wang, Yi Miao, Yulin Chen, Jatinder Singh, Ashish Sharma, Debanond Chakraborty, Deepak Kini, Yaser El-Sayed, and postdocs Dr. Sibsankar Haldar and Dr. Alagarsamy for all the discussions and making my stay in St. John's a pleasant one.

Last but not the least, my friends Rahman, Feroz, Animesh, Raghu, Venu, Varun, Ashfaq, Saif, and Mrs. Frances Power who were always there whenever I needed them.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

In traditional database systems, users interact with the database through transactions which are assumed to be Atomic, Consistent, Isolated and Durable, abbreviated as satisfying ACID properties [8, 18]. Specifically, *atomicity* refers to the *all or nothing* property, namely, that either all the operations of a transaction are executed or none of them is executed. That is, a transaction is to be treated as a single unit of execution. *Consistency* requires a transaction to be correct, that is, if executed alone the transaction will take the database from one consistent state to another. *Isolation* property does not allow transactions to read intermediate results of other transactions, that is, each transaction should observe a consistent database. Finally, *durability* requires the results of a committed transaction to be made permanent in the database in spite of possible failures. The ACID properties are ensured using two different protocols, one that ensures *execution atomicity* and the other which ensures *failure atomicity*. Execution atomicity refers to the problem of ensuring the consistency and isolation properties even when the transactions are executed concurrently. Protocols ensuring execution atomicity are referred to as *concurrency control* protocols. On the other hand, failure atomicity ensures atomicity and durability properties. Protocols

that ensure failure atomicity are called *recovery* protocols. Both these protocols are discussed in depth in [8].

With the emerging trend of database use in advanced applications like software development, book writing, office automation, CAD/CAM databases, cooperative applications, control flow systems, etc., the traditional transaction ACID properties seem either inappropriate or too restrictive and need to be relaxed as per application requirements. Some of the advanced transaction models presented in the literature addressing this issue are: Nested Transactions [42], Sagas [21], Acta [15], Cooperative Transaction Hierarchy [44], ConTract Model [52], Cooperative SEE (Software Engineering Environments) Transactions [18], Split-Transactions [49], and Flex Transaction Model [18].

The introduction of computer networks [62] led to distributed database technology which became one of the most important developments of the eighties. A *distributed database* is a database where objects may be stored at different sites and users may issue transactions at any site in the system [8, 61]. Further, the data objects may be replicated across sites to increase availability. Special replica control protocols have been designed to manage copies of data items across the network [35, 63]. In addition to homogeneous database systems connected across the network, integration of multiple, heterogeneous database systems was researched and proper consistency criteria for transaction execution proposed [7, 8, 9, 18, 23, 53, 57, 59, 61].

The nineties have seen advances in both wireless communication, which promise an ubiquitous computing environment, and handheld computers like palmtops and laptops which allow the user to carry his/her work anywhere. It is predicted that by the new millennium, tens of millions of users will be able to have access to information

from anywhere at any time. These emerging trends in wireless communications and hardware will change the way we compute and communicate. Some of the application areas are: emergency services (fire engines, accident reporting, ambulances, etc.), mobile users (news reporters, businessmen, traders, tourists), weather reporters (hurricane watches), banking applications (making purchases and booking tickets while travelling), military, transport (air, road), traffic regulations, etc..

Wireless networks introduce a new dimension in transaction processing. The limited execution capabilities of the mobile devices, varying degrees of connection (frequent disconnections, weak connections, occasional strong connections, and asymmetric connection), the relatively high costs in wireless access, and mobility of the devices raise a number of new research issues in the processing of transactions. Several new transaction models have been proposed addressing such issues. They are reviewed in Chapter 3.

1.1 Gist of the Thesis

In this thesis, we propose a new transaction execution model [39] for mobile environments. Due to the limited execution capability and memory restrictions of the mobile devices, and varying degrees of connection, the concurrency control mechanism most appropriate for mobile computation seems to be the optimistic approach. In the optimistic approach [34], the transactions are allowed to access data concurrently without any restrictions and the validation of the data read by the transaction is done only when the transaction comes for the final commit. Any modifications to the data items are done locally on a private copy and only after successful validation at the end are they made global. If validation fails, the transaction is rolled back

and restarted. More explicitly, in a mobile environment the mobile devices, while connected to the fixed network, hoard data required for continuing the computation in the disconnected mode in a cache. While disconnected, the computation is done at the mobile host using cached data, and later validated and committed at the end in the fixed network. Mobile transactions tend to get longer due first to the mobility of both data sources and data consumers, second to their interactive nature, i.e., pause for input from user, and thirdly to the frequent and unpredictable periods of disconnection. Thus, when the transaction comes for the final commit, there is a high possibility that the data it has read has become stale and hence it gets invalidated and aborted. In this thesis, we propose a transaction execution model that decreases the possibility of invalidations at the end of the transaction execution. We do this by validating (partial) computation at various intermediate stages, and "adjusting" the computation, if required, to some extent. This approach not only keeps the computation at the mobile devices in accordance with database changes on the fixed network, but also increases the possibility of the long running transaction to commit successfully.

Due to the inherent nature of the optimistic approach, despite frequent validations and re-executions at intermediate stages of the transaction execution, the transaction could still get aborted when it comes for the final commit. That is, the mobile host's computation is not guaranteed until it gets committed on the fixed network. We facilitate further reduction in the possibility of transaction abort by providing partial guarantees against invalidation. This is done by moving the computation towards pessimistic mode by "locking" some data items. A flexible concurrency control scheme which integrates optimistic and pessimistic approaches to access data items

based on Read/Write and Write/Write-conflicts is employed. With this approach, the transaction execution at the mobile host is made more credible.

1.2 Outline of the Thesis

Chapter 2 describes the mobile computing environment, the issues introduced henceforth and the general transaction processing mechanisms in the target environment. Chapter 3 surveys different mobile transaction models that have been proposed earlier. Chapter 4 introduces the need for validation and re-execution at intermediate stages of the transaction execution and provides a transaction execution model to achieve the same. Chapter 5 extends the proposed model by providing a concurrency control mechanism that integrates both the optimistic and pessimistic approaches to access data items. A detailed algorithm and the correctness proof follow. Chapter 6 concludes the thesis.

Chapter 2

CHARACTERISTICS OF MOBILE COMPUTING

The rapid advancements in wireless communications and its introduction in the distributed network are making it possible for mobile users to continue with computation from anywhere at any time. Still, the size and weight (limited resources, memory, and computational power), battery power (a finite energy source) and ergonomics of the mobile devices like laptops and palmtops impose several restrictions to this new paradigm which in the current literature is termed as *mobile* or *nomadic computing*. The applications and software support for mobile computing are still in the germinating stage due to the fact that the parameters to be taken into account are not yet clearly understood and defined. Furthermore, the data and computation mobility due to mobile host's movement, variable bandwidth, and frequent disconnections pose new challenges in the design of mobile applications and the processing of transactions in general.

Öszu and Valduriez [61] have provided an excellent classification for distributed DBMSs based on system autonomy, distribution and heterogeneity. Margaret Dunham et al. [17] extended this classification further to accommodate mobile database

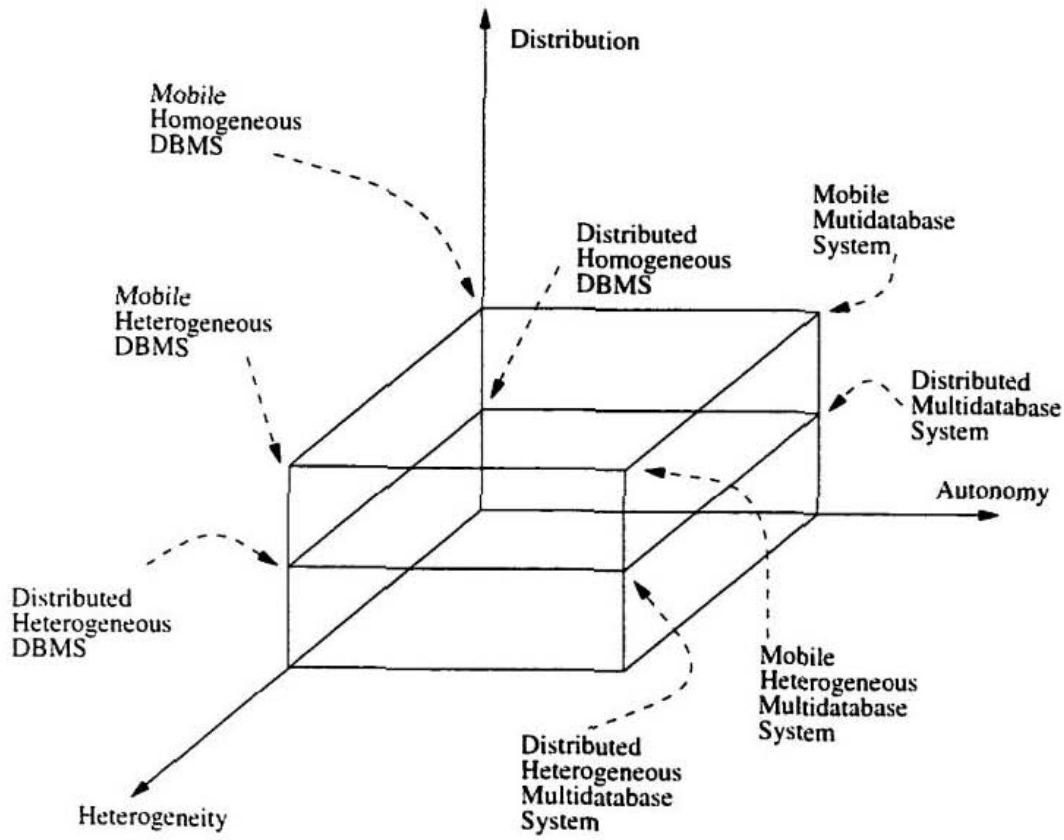


Figure 2.1: A Classification of Mobile Database Systems

systems by adding an extra point on the distribution axis as shown in the Figure 2.1. Therefore the mobile computing system can be viewed as a dynamic distributed system where links between nodes in the network change dynamically. These dynamic or changing links represent the connection between the mobile units and base stations to which they are connected as they move in the distributed system connecting to different base stations in the process.

This new classification has given a clear distinction for researchers to pursue their goals in definite areas of interest. In this thesis, I restrict the discussion to a Mobile

Homogeneous Database Management System.

2.1 The Mobile Database System

The architecture of a general mobile database system that supports mobile computing [4, 5, 20, 48, 32, 55] is shown in Figure 2.2. It consists of a static backbone network called the *fixed network*, a *wireless network*, *mobile hosts* (MH), and *mobile support stations* (MSS). A *host* that can move freely while retaining its network connection through the wireless network is a *mobile host* or a mobile unit. Hosts that are connected to the fixed network but unable to connect directly (due to lack of wireless capabilities) to the mobile host are referred to as *fixed hosts*. An MSS is a host that is connected to the static network through wired communication links, and is augmented with a wireless interface for the mobile host to interact with the static network. The MSSs are also known as *Base Stations*. Each MSS's wireless interface has a geographical coverage area called a *cell*.

2.1.1 Wireless Communication

The MHs require wireless access, although at times they may be able to connect to the wired network while stationary or at a desk for better and cheaper connection. The wireless interface can be either a Cellular Network which can offer a bandwidth in the order of 10 to 20Kbps *or* a Wireless Local Area Network (LAN) which can offer bandwidth in the order of 10Mbps (e.g., NCR Wavelan, Motorola ALTAIR). The fixed wired networks on the other hand can offer bandwidths in the order of 10Mbps for Ethernet, upto 100Mbps for FDDI and 144Mbps for ATMs. Though these numbers may change in the future, it is safe to assume that the bandwidth will remain a major

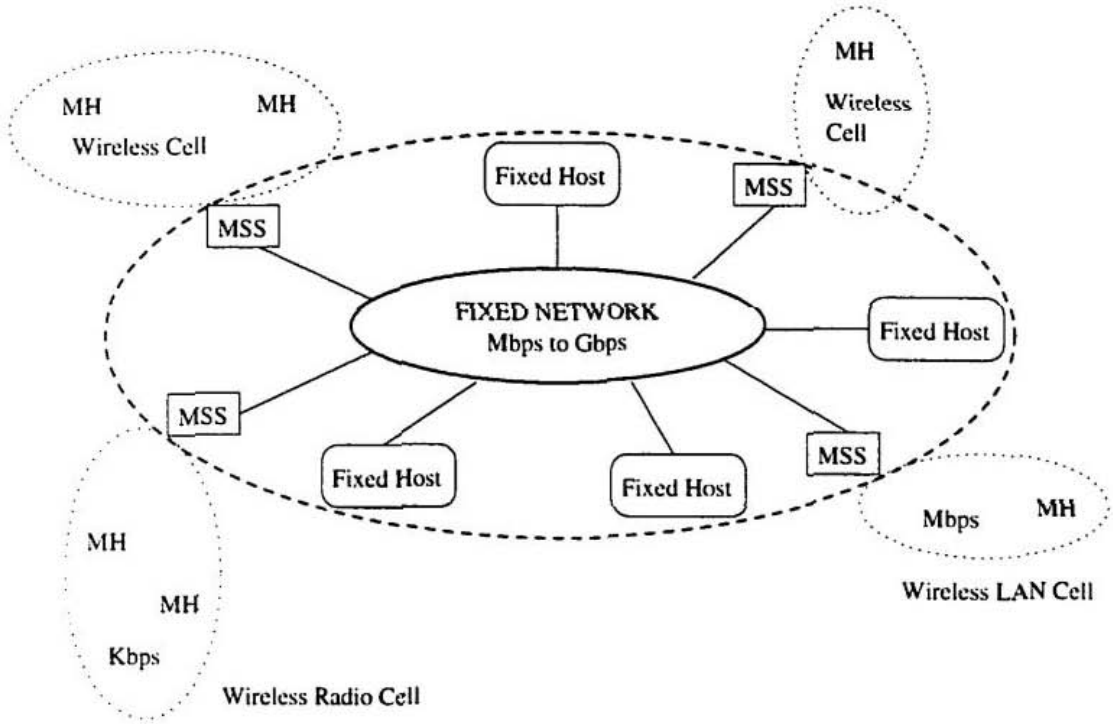


Figure 2.2: Mobile Database System

bottleneck and would limit the performance for mobile database system design. The design challenges resulting from the wireless communications are :

1. *Frequent Disconnections*: To reduce the costs incurred due to wireless network access, power consumption, or bandwidth use, the MHs are often disconnected from the network. Disconnection here implies a voluntary disconnection and not a failure. And so, a MH informs the system of an impending disconnection prior to its occurrence. Both disconnection and subsequent reconnection of a MH can be initiated *only* from the MH: it is isolated from the system in the intervening period. *Disconnected operation* would then imply the ability of the Mobile Hosts to operate despite server accessibility by emulating server services

locally.

2. *Weak and Variant Connectivity*: Wireless networks are more expensive, offer less bandwidth, and are less reliable than wired networks [48]. A MH may be connected to different networks at different times, with varying degrees of bandwidth and reliability. The weakest connection is, of course, the disconnection.

3. *Broadcast capability*: The communication channel between the MH and the MSS is asymmetric. There exists a high bandwidth broadcast channel from the MSS to all the MHs in the MSS's cell [71]. This can be advantageous in disseminating (broadcasting) updates or information to a group of MH's. Both push and pull based techniques are re-visited [2] to make the best use of broadcast capabilities.

4. *Tariffs*: There could be a cost on the MH's users based on either the amount of connection time used or the number of messages passed using the wireless network.

5. *Security Risks*: Since it is easier to connect to a wireless link, the security of wireless communication can be compromised much more easily than wired communication, especially if the transmission range encompasses a large area.

2.1.2 Mobility

The ability to move allows the MHs to connect to the fixed network from different points over time. Thus, the system configuration is not static anymore. The distributed algorithms which rely on the fixed topology have to be re-designed to accommodate the movement of MHs. The amount of bandwidth available and the load on the MSSs change with time depending on the number of devices currently in

the network cell supported by the respective MSSs. Another problem introduced by mobility is *Address Migration*. As the MHs move, they use different network access points (or addresses) in the network. Current networking is not designed to change these addresses dynamically. If a MH needs to be reached, messages must be sent to its most recent address. There are four basic mechanisms to determine the current address of the MH: broadcast, central services, home bases, and forwarding pointers. These are the building blocks of the current proposals for the 'mobile-IP' schemes [60].

When a MH moves into a new cell, three steps need to be taken care of: (1) termination of the communication with the current MSS; (2) establishment of communication with the new MSS; and (3) changing the network routing to reflect the new base station. This process is known as *hand-off* (or *hand-over*) and it may result in loss, duplication and disordering of packets which degrades the performance of the transport protocol. Additional costs are incurred in locating a MH in the distributed network and need for efficient addressing algorithms arises due to mobility of the MHs.

2.1.3 Portability

Desktop computers are not intended to be carried thus making their design easier in terms of weight, power, space, cabling and heat dissipation. The design of MHs on the other hand should strive for the properties of a wrist watch: small, lightweight, durable, water-resistant, and long battery life. Concessions can be made in these areas to increase or enhance the functionality for different requirements. But finally the 'value' provided to the user must exceed the trouble of carrying the device. Considerations such as small and lightweight MHs, in conjunction with a given cost and

level of technology, will keep MHs having less resources than static elements, including memory, screen size, and disk capacity. Furthermore, MHs rely on a finite energy source (batteries) for their operation. This concern for power will remain even with advances in battery technology and would require to re-visit design in software and hardware technologies. Finally MHs are more susceptible to accidental damage, being stolen, or being lost.

2.2 Transaction Processing in Mobile Database Systems

The limited resources available on the mobile host, frequent disconnections, and mobility of the hosts force us to revisit the concurrency control methods developed for traditional database systems. While the introduction of mobility in the distributed environment poses new issues relating to processing of a transaction across several mobile support stations, the frequent disconnections of the mobile host leads to transactions which are long running. In general, the mobile transaction processing may be structured in one of the three ways [43] as described below:

2.2.1 Mobile Host as I/O Device

Due to the weight and size restrictions, the MH may have very few resources available on it, thus indicating a very slow CPU and little memory. Consequently, the MHs can only run small programs. Programs like database systems cannot fit into its memory. Then, all the data and issues relating to the execution and management of transactions acting on this data reside in the static part (MSS) of the network. In this structure, the MH does not do any computation. It simply submits the operations of

the transaction to the MSS. The MSS executes the steps and sends the results to the MH. The main disadvantage in this mode of operation is that transaction execution is possible only when the MH is connected to the MSS. Also, due to communication over the slow, low bandwidth network, the response time of the operations of the transaction is increased.

2.2.2 Complete Database Server on the Mobile Hosts

Mobile Hosts on the other hand may have a high speed CPU along with relatively Large storage capabilities. This enables us to place data locally and run a big program like a database server to manage transactions on the MH itself. This appears to be a good idea if the (subset of) data kept on the mobile host is accessed only by the MH's user, that is, locally. In a way, this implies that the (subset of) data placed on the MH is basically checked-out by the MH. Thus, the other MHs and MSSs in the network cannot access this part of the database. This structure improves the response time of a particular user, though at the expense of others. This is generally not acceptable. An exception to this is queries and updates to the database that are location specific. Here, we know that most operations on the data will be from the local user.

One important issue to be considered in this case is the overhead of keeping the data on the MHs consistent with the data on the MSS. This again depends on the kind of consistency guarantees the database system makes and the duration of disconnection of the MH from the network. In some databases, some of the ACID (Atomicity, Consistency, Isolation, Durability) constraints of transactions may be relaxed to give more flexibility to transaction management on the MHs.

This mode is not feasible since in most cases (for example, in satellites and traveler's notebooks [32]), the size and weight of a MH can only be small and this imposes serious restrictions to the computer's memory as well as the variety of devices included in the computer.

2.2.3 Mobile Hosts with Cached Data

As the reads/writes ratio becomes smaller, a structure in between the above two appears appropriate. For, if the number of writes increases, in the second method, we will require a constant connection with the fixed network to transfer the updates into the system immediately. Again, we can argue that if we have continuous connection (as in the first case), the processing power on the MH is not fully utilized. This new structure still keeps the (subset of) data on the MH locally, but now this data is treated as a *cache* rather than a primary copy. Before disconnection, the MH is allowed to cache the (subset of) data required for continuing the execution of transactions. The responsibility of maintaining consistency between the primary copies on the MSS and the cached data on the MH is given to the MSS. The transactions are processed locally on the MH. At the time of commitment, the MSS is contacted and requested to try and commit the transaction. If the transaction aborts, the MSS sends a message to the user at the MH, and the user can redo the transaction on new cached data. This seems to be the right approach overall since we are moving the expensive part of the transaction processing to the static portion of the network, where communication is an order of magnitude cheaper.

2.3 Summary

In this chapter, the mobile computing environment has been presented and some of the fundamental challenges due to wireless networks, frequent disconnections, and mobility of the hosts in the distributed network has been discussed. In addition, some basic methods in processing of transactions in disconnected environment have been discussed.

Chapter 3

LITERATURE SURVEY

The restrictions imposed by wireless medium, frequent and unpredictable periods of disconnection, mobility of the users, computing power and battery life require the mobile transactions to have the following characteristics which are different as compared to traditional transactions [5, 6, 14, 17, 20, 32, 38, 43]:

1. The mobile transactions might have to be structured as a set of transactions some of which execute on mobile hosts while others execute on the mobile support station. The mobile transaction might share its partial results with other transactions due to disconnection and mobility.

2. The mobile transactions require computations, state of computations, queued requests, and communications to be supported by both mobile hosts and mobile support stations.

3. The mobile transactions require the mobile support stations to support the hand-off process as mentioned in the previous chapter when mobile hosts move from one cell to another.

4. The mobile transactions tend to be long-lived due to:

- the mobility of both data sources and data consumers (hand-off).
- their interactive nature, i.e., pause for input from user.
- the frequent and unpredictable periods of disconnection.

5. The mobile transactions should support and handle concurrency, recovery, disconnections, cache coherency and mutual consistency of the replicated data objects.

There has been tremendous research focused on transaction processing in mobile computing environments in the last decade to accommodate these characteristics. Some of the techniques suggested are reviewed in the following sections.

3.1 Optimistic Models

The optimistic concurrency control scheme proposed in Coda file system [30] allows the cached objects in the mobile host to be updated without any co-ordination but the updates need to be propagated and validated at the database servers for the commitment of transactions. The central idea behind [30] is that *caching of data*, now widely used to improve performance, can also be exploited to enhance *availability*. The key mechanisms for supporting disconnected operation includes three states: *hoarding*, *emulation* and *reintegration* as shown in the Figure 3.1. Venus, the client cache manager, while in hoarding state, relies on server replication but is always on the alert for possible disconnection and ensures that critical objects are cached at the moment of disconnection. Upon disconnection, it enters the emulation state. During the disconnection period it remains in the emulation state and relies solely on the

contents of its cache. It records sufficient information to replay update activity with extensive optimizations and reintegrates upon reconnection.

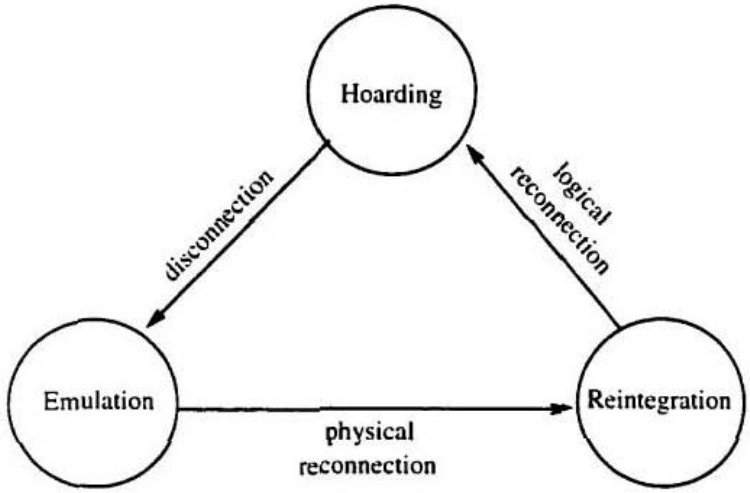


Figure 3.1: Venus states and transitions

The scheme leads to aborts of mobile transactions unless the write-write conflicts are rare. Since mobile transactions are expected to be long-lived due to disconnection and long network delays, the conflicts will be more in a mobile computing environment.

Isolation-only transactions (IOTs) proposed in [36, 37], is an instance of the open nested transaction model in [14] and an extension to Coda which uses serializability constraints to automatically detect read/write conflicts. The model supports a variety of conflict resolution mechanisms (which might employ application semantics) for automatic conflict detection and resolution leaving manual resolution as a last resort. IOTs are a sequence of file access operations. Transaction execution is performed entirely on the client and no partial result is visible on the servers. *First class* transactions are those transactions that have no partitioned file access (i.e., the client

machine maintains a server connection for every file the transaction has accessed) are committed immediately on the servers. A *second class* transaction T on the other hand has partitioned file access and its results are maintained in the local cache and visible only to subsequent accesses of the same client. A first class transaction is guaranteed to be *serializable* with all previously committed or resolved transactions at the server. On the other hand, a second class transaction is guaranteed to be locally serializable among themselves and is in a pending state until reconnection time. On reconnection the pending transaction has to be validated against one of the two proposed serialization constraints. The first criterion is global serializability, which means that if a pending transaction's local result were written to the servers *as is*, it would be serializable with all previously committed or resolved transactions. The second criteria offers stronger consistency and is called global certifiability (GC) which requires a pending transaction be globally serializable not only *with*, but also *after*, all previously committed or resolved transactions [36]. Intuitively, GC assures that the data accessed by a pending transaction are unchanged on the servers between the start and the validation of the transaction. Transaction validation failures are resolved by aborting the transaction, re-executing it, invoking a user specified application specific resolver (ASR), or, as a last resort (default option), the inconsistencies are exposed to the user for manual resolutions (or repair).

A new *two-tier* replication algorithm is proposed by Gray et al. in [24] to alleviate the unstable behaviour observed in the *update anywhere-anytime-anyway* transactional replication scheme when the workload scales up. Lazy master replication that is employed in the algorithm assigns an owner to each object. The owner stores the object's correct current value. Updates are first done by the owner and then prop-

agated to other replicas. The two-tier scheme assumes two kinds of nodes: *mobile nodes* and *base nodes*. Mobile nodes are often disconnected and store a replica of the database. The mobile nodes accumulate tentative transactions that run against the tentative database stored at the node. The base nodes on the other hand are always connected and store replicas of the database. Each object is mastered at some node, either the mobile node or the base node. When the mobile node reconnects to the base station, it sends any replica updates mastered at the mobile node to the base node, sends the tentative transactions (and all their input parameters) to the base node to be re-executed as base transactions on the master version of data objects maintained at the base nodes in the order in which they tentatively committed on the mobile node. The base transaction has an *acceptance criterion*: a test the resulting outputs must pass for the slightly different base transaction results to be acceptable. If the base transaction fails its acceptance criteria, the base transaction is aborted and a diagnostic message is returned to the user of the mobile node. While the transactions executed on objects mastered on the mobile nodes are confirmed, those executed on the tentative object versions have to be checked with nodes that hold the master version. The key properties of the two-tier replication scheme are: (1) mobile nodes are allowed to make tentative updates while being disconnected, (2) base transactions execute with single-copy serializability so the master base system state is the result of a serializable execution, (3) a transaction becomes durable when the base transaction completes, and (4) replicas at all connected nodes converge to the base system state.

3.2 Semantic Models

Chrysanthis and Walborn in [14, 68] extend the semantic based transaction processing scheme to the mobile computing environment to increase concurrency by exploiting commutative operations. This extended model views mobile transaction processing as a concurrency and cache coherency problem. The model assumes the mobile transaction to be a long lived one characterized by long network delays and unpredictable disconnections. This approach utilizes the object organization to *split* large and complex objects into smaller manageable fragments thus allowing several users to access the object concurrently. The mobile hosts specify the granularity and usage constraints of an object to be cached by using the split operation. The split request from the mobile hosts consists of two parameters: *selection criteria* and *consistency conditions*. The selection criteria specify the object to be cached and the required size of the object partition. The consistency conditions specify constraints on the fragment which need to be satisfied to maintain the consistency of the entire object. The server dishes out the fragments of an object as requested by the MH. On completion of the transaction the MH returns the fragments to the server. These fragments are again put together by the *merge* operation on the server. If the fragments of the object can be recombined in any order to reflect an alternative sequence of operations on the object then the objects are termed *reorderable* objects. And objects that can be extended by these two operations (split and merge) are referred to as *fragmentable objects*. Aggregate items, sets, and data structures like stacks and queues are examples of fragmentable objects.

Another semantic model has been proposed by Klingemann et al. [31] wherein workgroup computing or cooperative work has been given focus in the aspect of

mobile computing. The issue in question here is that most of the models or approaches [30] to support disconnected operation rely on the assumption that the degree of data-sharing is low which is obviously not appropriate for cooperative work. CoAct cooperative transaction model [54, 69] has been utilized and extended to provide support for parallel, disconnected activities. The model incorporates the notion of resolvable, simultaneous work. In general, cooperative work is characterized by alternative periods of individual and joint work. The CoAct model assigns a *private workspace* to every user who takes part in a *cooperative activity*. By default, the private workspaces of the co-workers are isolated from each other. In addition, there exists a *common workspace* for each cooperative activity. All the users involved in the activity would then integrate their relevant contributions into the common workspace such that there is a single result of the cooperative activity. A cooperative activity is described by (1) a set of operations that can be invoked by a user in his private workspace (the sequence of operations executed is maintained in a *workspace history*), and (2) a set of type-specific merging rules that exploit the semantics of operations to guide the process of information exchange (*history merging*) [69]. Co-workers are allowed to exchange operations between their private workspaces by means of *import* and *delegate*. They can exchange operations through the common workspace by means of *save* and *import*. All this information exchange is done through history merging which is the core technique. The flexibility of history merging is mainly achieved by its ability to dynamically determine consistent units of work in terms of operations and its consideration of operation semantics for resolving conflicts. Furthermore, the merge process need not be atomic in the sense that all or none of the operations to be executed are incorporated into the destination workspace. It is pos-

sible to partition the subhistory to be exchanged to so called independent histories which can be exchanged independently. This enables a finer granularity of conflict resolution [69]. While traditional transaction processing guarantees that no errors occur due to the interleaved execution of transaction (serializability), the correctness criteria of the merge approach guarantees that no inconsistencies are introduced due to the exchange of information between concurrently executed work.

3.3 Object Oriented Models

Replication of data on the mobile host as a cache is one of the solutions to solve the problem of uncertain availability due to frequent disconnections. Traditionally, replicated database replica control takes care of maintaining the consistency among replicas of a data item. The replica control also determines whether to let a transaction proceed or not using replica control protocol (quorum consensus, primary copy, etc.). In the extended architecture presented by Rasheed and Zaslavsky [51], there is a logical replicated database (LDB) which contains the objects, some (or all) of which can be replicated. Each *object* in the LDB is a *collection* called *meta-object*, which provides one view for all the replicas of a data item. The collection of replicas of a data item is encapsulated within this meta object which is responsible for the consistency control among different items in its collection. For this purpose *object replica control* layer as shown in Figure 3.2 is built within the meta object and is essentially a method of it.

If the number of encapsulated replicas in the meta-object is one then there is a one-to-one mapping between logical replicated database and physical replicated database. The concept of *twin-transactions* [50] is utilized to replicate the processing of a trans-

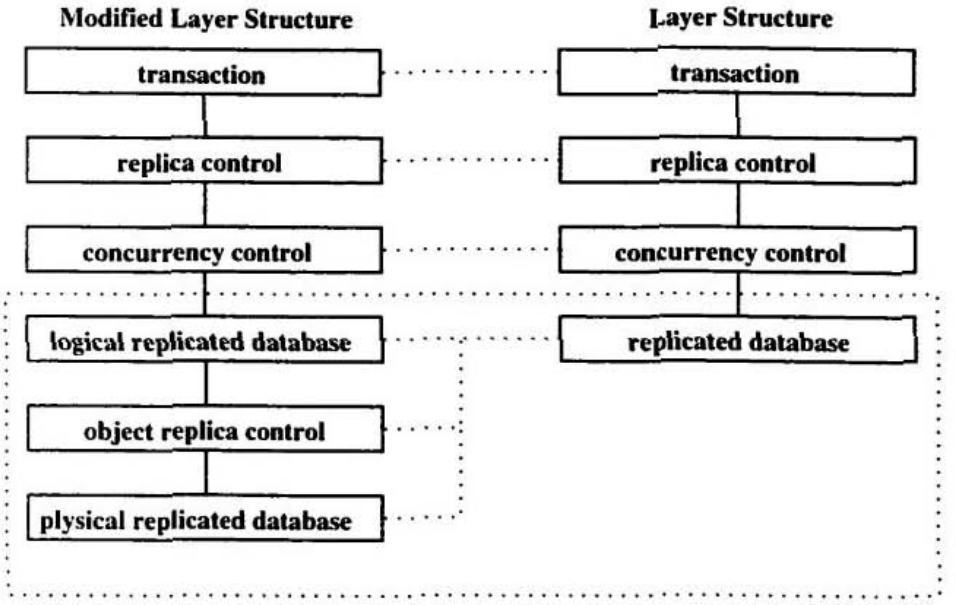


Figure 3.2: Abstract model of the Multi-Layer Architecture

action and to guarantee successful completion of a transaction, even in the case of disconnections of certain replicas, with a certain probability of success. Specifically, when a replicated data item receives a transaction T, it will be mirrored to generate twin-transactions T_r, T_o. While the twin-transaction T_r will execute on the data item maintained by the host receiving the transaction T, the other twin-transaction T_o will be sent to the meta-object O_{meta}. If O_{meta} is not available at the moment, then the twin-transaction T_o will be kept in a history log and transmitted whenever O_{meta} becomes available. The meta-object O_{meta} will eventually receive transactions from different hosts maintaining replicas of data item X and is responsible to resynchronize all the replicas.

The Thor distributed object-oriented database [25] provides a persistent universe of typed, encapsulated objects. Computations take place within atomic transactions

that typically make use of many objects. Disconnected operation in such a system results in new challenges because of the small size of objects (compared to files), the richness and complexity of their interconnections, the huge number of them being accessed and that too within atomic transactions. Applications interact with Thor by invoking object methods and requesting transaction commits. Getting the right objects in the cache or hoarding before disconnection is taken care by database query language (object-oriented query languages). While disconnected from the system, the clients use dependent commits to tentatively commit the transactions. Transactions are synchronized using optimistic concurrency control. A portion of the Thor called the frontend (FE) keeps track of the persistent objects the transaction reads or writes. When the client requests for a commit, the FE communicates to a server (one containing some used objects) that acts as a coordinator of a 2-phase commit. Transaction is validated in the first phase by all the participating nodes. If all participants validate the transaction, the coordinator commits it, else the transaction is aborted. In either case, the coordinator informs the user; it notifies participants of its decision in the background. Further, high-level semantics of objects is used to avoid transaction aborts.

Pitoura and Bhargava in [46] present a flexible two-level consistency model to deal with the frequent, foreseeable and variable disconnections that occur in a mobile computing environment. Closely located and semantically related data are grouped together to form a *cluster*. While all the data inside the cluster are mutually consistent, degrees of inconsistency are allowed between data of different clusters. The object clustering is dynamic and the transactions are allowed to exhibit certain degrees of tolerance for inconsistencies by using weak-read, weak-write, strict-read and

strict write. Strict-read and strict-write have the same semantics as normal read and write operations invoked by transactions satisfying ACID properties [8]. Weak operations are operations that can be executed under weaker consistency requirements. A weak-read returns the value of a locally cached object written by a strict-write or a weak-write. A weak-write updates the local copy, which might become permanent on cluster merging if the weak-write does not conflict with any strict-read or strict-write operation. Furthermore, the mobile transaction is decomposed into a set of weak and strict transactions. The decomposition is done based on the consistency requirement.

Joseph et al. in [28] have proposed the Rover software toolkit that offers applications a client/server distributed object system with client caching and optimistic concurrency control. Clients are Rover applications that typically run on mobile hosts, but could be run on stationary hosts as well. Servers, which may be replicated, typically run on stationary hosts and hold long term state of the system. The toolkit supports a set of programming and communication abstractions that enable the construction of both *mobile-transparent* and *mobile-aware* applications. The objective of the mobile-transparent approach is to develop proxies for system services that hide the mobile characteristics of the applications. Since the applications can be run without alteration, this approach is appealing. However, to excel, the applications must often be aware of and take active part in mitigating the harsh conditions of a mobile environment.

The Rover toolkit provides mobile communication support based on two concepts: *relocatable dynamic objects* (RDOs) and *queued remote procedure calls* (QRPCs). An RDO is an object encapsulating both code and data with a well defined interface that can be dynamically loaded into a client computer from a server computer, or vice

versa, to reduce client-server communication requirements. All application code and application data are written as RDOs. QRPC is a communication system that permits applications to continue to make non-blocking remote procedure calls [10] even when a mobile host is disconnected. That is, all the requests are stored in a local stable log and control is immediately returned to the application. The requests and responses are exchanged upon network reconnection. The main task of the programmer when building a mobile-aware application with Rover is to define RDOs for the datatypes manipulated by the application, and for the data transported between client and server. The programmer then divides the program into portions that run on the client and portions that run on the server: these parts communicate by means of QRPCs. The programmer then defines methods that update objects, including code for conflict detection and resolution. Then, the modules that compose the client and server portions of an application are linked using the Rover toolkit. The application can then actively cooperate with the runtime system to *import* objects onto the local machine, *invoke* well-defined methods on those objects, *export* logs of method invocations on those objects to servers, and *reconcile* the client's copies of the objects with the server's.

Walborn and Chrysanthis in [67] proposed PRO-MOTION, a mobile transaction processing infrastructure that supports disconnected transaction processing in a mobile client-server environment. The fundamental building block of this architecture is the *compact*, the unit of caching and replication. When a wireless client needs data, it sends a request to the database server. The server sends a compact as a reply. A *compact* is an object that encapsulates the cached data, operations for accessing the cached data, state information (such as number of accesses to the object), con-

sistency rules that must be followed to guarantee consistency, and obligations (such as deadlines which creates a bound on the time for which the rights to a resource are held by the mobile host or restrictions on the visibility of locally committed updates). Compacts provide flexibility in choosing consistency methods from simple check-in/check-out pessimistic schemes to complex optimistic criteria. If the database server lacks compact management capabilities, a compact manager acts as a front-end to the database server. PRO-MOTION consists of four transaction processing activities: hoarding - the mobile host is connected to the network and stores compacts in preparation for an eventual disconnection, connected processing - the mobile host is connected to the server and the compact manager is processing the transactions, disconnected processing - the mobile host is disconnected from the network and the compact manager is processing transactions locally, and resynchronization - the mobile host is reconnected to the network and the updates committed during disconnection are reconciled with the fixed database.

3.4 Mobility

Compared to distributed transactions, mobile transactions do not originate and end at the same site. *Kangaroo transaction model* proposed in [17] captures both the data and movement behavior. While mobile behavior is realized via the use of *split* transactions [29, 49], data access behavior is captured using the concepts of *global* and *local* [8, 9, 11, 13, 18, 22, 61] transactions in a multidatabase system. Data on the source system(s) is accessed by the mobile transaction through the *Data Access Agent* (DAA). The DAA is responsible for controlling the mobile transaction on the base station and maintaining necessary logs. Each subtransaction represents the unit

of execution at one base station and is called a *Joey Transaction* (JT). When the mobile unit hops from one cell to another, the control of the Kangaroo Transaction (KT) changes to a new DAA at another base station. The DAA at the new base station creates a new JT (or subtransaction as part of the hand-off process) using the split operation. A doubly linked list maintains the order of different JTs executed by the mobile transaction. The KT model may operate in *compensating* or *split modes*. While operating in compensating mode, the failure of a JT causes the entire KT to be undone. This is accomplished by compensating the previously completed JT's. The split mode on the other hand is the default mode. If a JT fails in this mode no new global or local transactions are requested as part of the KT and the previously committed JTs are not compensated for. Neither of these modes guarantees the serializability of the kangaroo transaction.

A weakly replicated system is characterized by the lazy propagation of updates between servers and hence the possibility of mobile hosts to see inconsistent values when reading data from different replicas as they move across servers. In effect, a user may read some value of a data item and then later read an older value *or* a user may update some data item based on reading some other data, while others read the updated item without seeing the data on which it is based on. Four per-session guarantees are introduced in [64] to alleviate the problems observed in weakly consistent systems [58] while maintaining the principle advantages of read-any/write-any replication. These session guarantees were developed in the context of the Bayou project at Xerox PARC [16], to reduce client-observed inconsistencies when accessing different servers. A *session* is an abstraction of read and write operations performed during the execution of an application. In contrast to atomic transactions that ensure

both atomicity and serializability, the intent of a session is to present individual applications with a view of the replicated database that is consistent with their own Reads and Writes performed in the session even though these operations are directed to various, potentially inconsistent servers. In brief, the guarantees are: *Read Your Writes* - read operations reflect previous writes, *Monotonic Reads* - successive reads reflect a non-decreasing set of writes, *Writes Follow Reads* - writes are propagated after reads on which they depend, and *Monotonic Writes* - writes are propagated after writes that logically precede them. These properties are guaranteed in the sense that either the storage system ensures them for each read and write operation belonging to a session, or else it informs the calling application that the guarantee cannot be met.

3.5 Other approaches

Yeo and Zaslavsky in [70] have presented a basic architectural framework to manage mobile transactions in multidatabase systems. A major premise of this architecture is that the users of the mobile units may voluntarily disconnect from the network prior to their submitted global transactions being completed. The coordinating site can then schedule and coordinate the execution of the global transaction on behalf of the mobile workstation. A simple *message* and *queuing facility* is suggested which provides a common communication and data exchange protocol to effectively manage global transactions. Transaction sub-queues are used to model the state of global transactions (based on the concept of finite state machines). While disconnected, the user of the mobile workstation can perform some other tasks thereby increasing processing parallelism and independence.

Madria and Bhargava in [38] incorporate pre-read, prewrite and precommit operations along with the original operations in the transaction execution [8]. Each mobile transaction has a prewrite operation before a write operation. The pre-write operation does not update the state of a data item but only makes visible the value that the data item will have after the commit of the transaction. A pre-read returns a pre-write value whereas a read returns a write value. Once all the prewrites have been processed, the mobile transaction pre-commits at mobile host. Precommitted transaction is guaranteed to commit and its results are visible by all other transactions on mobile and stationary hosts before the final commit thus increasing data availability.

In [41] Morton and Bukhres present a recovery strategy for transactions created by a mobile host in a distributed medical patient database environment. To ensure that transactions spend minimum time in recovery, save-points of pure forward recovery of Sagas [21] are utilized. The base station agent (BSA) proposed in the architecture is responsible for the execution and recovery of the transaction created by a mobile host, which is no longer connected to the mobile network. Saga, a transaction model for long-lived activities, consists of a set of independent (component) transactions $t_1, t_2, \ldots t_n$ which can interleave in anyway with component transactions of other sagas. The component transactions within a saga execute in a pre-defined order which, in the simplest case, is either sequential or parallel. The component transactions are able to commit without waiting for other component transactions or the saga to commit. In pure forward recovery, save-points are taken at the beginning of every subtransaction. Thus these save-points minimize the amount of recovery that a transaction must undergo. Only the failed subtransaction needs to be aborted and

resubmitted by the BSA. Here we should note that pure forward recovery methods would be useful for simple long lived transactions that always succeed.

Alonso et al. in [3] have explored Workflow Management Systems in the context of a mobile environment. The goal here is to give enough autonomy to the users (clients) to facilitate them to work independently without having to be connected to the rest of the system and still maintain the overall correctness and consistency of the processes being executed. The paper introduces the notion of *locked activities* in the workflow systems and the user's commitment to eventually execute them. The workflow activities are maintained in worklists for the users to execute. Typically a user on connection selects one or more activities from the worklist which he wishes to perform and removes it from the worklists of the other users (synchronization phase). The user may work on these locked activities in a disconnected mode (disconnected operation phase) and later resynchronize it with the ongoing workflow (reconnection phase).

Chapter 4

RE-EXECUTION MODEL

As seen earlier, mobile hosts can operate as simple input/output devices with very little computing capabilities. They can also be full-fledged nodes of the distributed database system. In practice, very few applications fall in either of these categories. The common trend is *some* computing capabilities and *some* memory in the mobile hosts. Mobile hosts are then expected to do the relevant computations and provide quick response to the users, but the task of preserving the traditional ACID properties of transaction is delegated to the fixed network which supports the mobile hosts. Thus the mobile environments introduce a new dimension in transaction execution.

In this chapter, a new transaction execution model for the mobile environments is proposed. It is designed to satisfy the dual requirements of quick computation as well as keeping the database consistent. The proposal builds upon various other proposals found in the literature, for example in [16, 24, 47, 56]. This model is helpful for recovery purposes also.

In the next section, we introduce the model and its usefulness with some examples. The model is described in detail in section 4.2, and section 4.3 discusses the suitability of the model for the various transaction processing issues in mobile environments.

4.1 Motivation

Among the three modes of transaction execution, discussed in the second chapter, the third one, namely, caching data and executing transactions on that data, in the MH, seems to be the most practical mode. It is widely accepted that the primary purpose of transaction execution at the MHs is to minimize the response time to users [43]. The limited capabilities of the MHs and the expectation that they be able to operate even in disconnected mode make it impossible for the MHs to satisfy the traditional ACID properties of the transactions. Therefore, the strategy that has been advanced in the literature is to execute transactions, to the possible extent, at MHs, but delegate the responsibility for their ACID properties to the MSS. This involves two tasks:

1. keeping the data values at the cache of the MH as far up-to-date as possible, and

2. keeping the MSS informed about the transactions executed at MH as quickly as possible.

In summary, the users of a mobile host need to have some idea of the validity of the data items they read, and some assurance that their updates will be saved at a later stage.

The simplest transactions that can be executed in MH are the read-only transactions which read data from the cache, do not have to produce up-to-date values, and do not have to be serialized with other transactions executed at MSS. These transactions can simply be ignored by the fixed network, when looking for serializability criteria. A cache for these purposes is called a *snapshot* in [25].

All other transactions executed at MH are 'checked' with MSS. The fact that the transactions may be executed at MH while it is disconnected from the MSS, and the executions will be checked with the MSS when MH gets connected to MSS, suggests naturally an optimistic scheme of execution. Optimistic concurrency control [34] necessitates validation of the transaction steps, that were executed at MH, at MSS with respect to the current data values and other transactions validated thus far.

The idea of re-execution of transactions at MSS, instead of just validation, has been proposed in several papers [16, 24, 47]. Pitoura and Bhargava [47] talk about *transaction proxies*: for each transaction executed at the MH, its dual transaction called proxy is defined and is executed on the MSS. The proxy transaction is a sub-transaction of the original transaction and contains only the updates. The execution at MH is to convey the results to the user, whereas the execution at the MSS is for recovery purposes. The vulnerability of MH to theft, loss or accidental destruction, and the unreliability of communication through wireless connection suggests that the MSS take this step for recovery purposes [47].

Transaction re-execution at MSS has been proposed in [16, 24] also: here, *not for recovery purposes but for the correctness of the results*, since the values of the data cached in the MH might have changed in the fixed network in the period of disconnection of MH. The transactions (or steps) executed on the cached data (tentative data) in MHs are referred to as *tentative transactions*, and they are re-executed on the MSS as *base transactions* [24]. Note that if the cached values had not changed at all in the fixed network, then validation itself will be successful and re-execution will not be needed. If the values did change, then validation will not be successful. In this case, instead of simply aborting the transaction, re-execution is opted. Re-execution may

not produce exactly the same results as of the original execution at MH. However, as long as the results of the execution at MSS satisfy certain acceptance criteria the execution is considered successful [24]. If the acceptance criteria are not met, then the user is consulted for possible abort or reconciliation. We illustrate this with an example.

Example 1 : In a stock exchange bureau, some of the stockbrokers may have to meet the clients personally. Before moving or on his way to the client location a stockbroker may cache the current share values of different companies along with some background information regarding the respective companies which would help the client to make decisions in buying the stocks. He may then disconnect, meet the client and show the current stock rates available on his disconnected MH. While in disconnected mode, his client may request the sale or purchase of some of the stocks, according to the values available on the MH. At this point, the stockbroker executes the transactions locally on the cached data values. Since the client knows that the stock prices change, he may also associate the amount of divergence allowed on the stock prices for the sale or purchase. On reconnection at a later time with the fixed network, the transaction is re-executed using the up-to-date stock prices and tested with the acceptance criteria. If the results are within the specified divergence, the transaction is committed on the MSS. Otherwise, the transaction is aborted and the user notified. While on his way to the next client's location, the stockbroker can refresh his cache with stock prices and information of other companies which the next client would be interested in as depicted in the Figure 4.1.

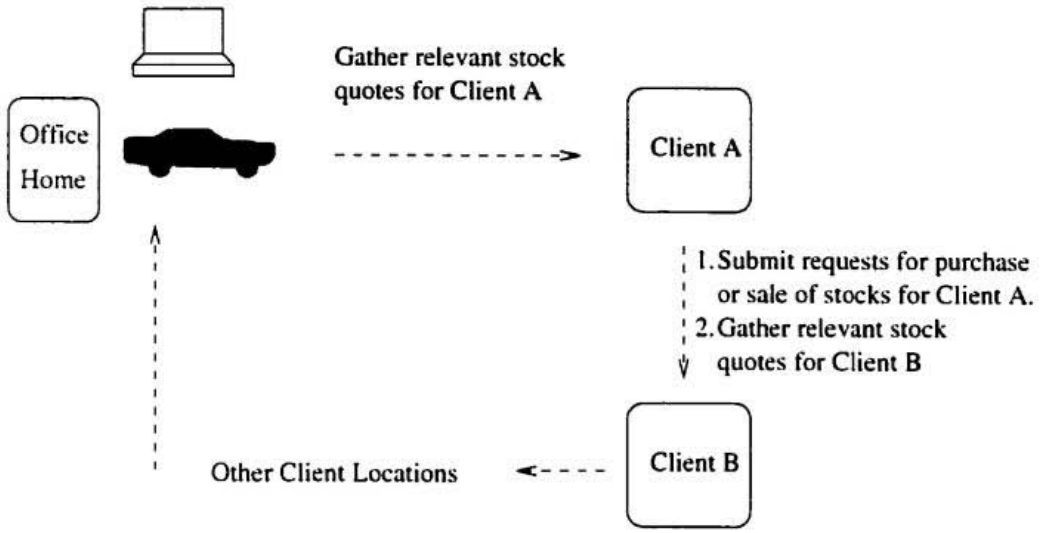


Figure 4.1: Stockbroker's movements on a typical working day

In the reconnection period, the MSS may also inform the state of the previous transactions submitted. The stockbroker can thus inform his previous clients (whom he had serviced earlier) using a cellular phone about the final results of their respective transactions while on his way to new client locations. □

Transaction Types: The transactions submitted by the user on the MH may be either *interactive* or *non-interactive*. A non-interactive transaction is submitted as a single request message by the user, whereas an interactive transaction is submitted as multiple request messages, or *steps*. Each step may consist of one single operation (example, read/write) or a group of operations. The interactive transaction tends to be long running as it might have to wait for user input from time to time. The frequent disconnections and unpredictable periods of disconnections further makes the transaction execution longer thus increasing the possibility of the transaction having

read stale data.

In this thesis, we propose *to extend the re-execution idea to various intermediate stages of execution of a transaction* at MH. At every stage, all the steps which have been executed at MH thus far are validated at MSS, and if validation fails, re-executed. The results of re-execution are checked against the acceptance criteria. Of course, this could be done only when the MH is able to connect to the MSS. Connection points (of MH with MSS) may define the stages. The disconnected periods, between consecutive connection points, may be of arbitrary duration, and therefore the number of steps executed in those periods may vary. The longer the disconnection period, it is more likely that the cached data values have changed, and so the re-execution results diverge from those obtained in MH. If the new results are not acceptable, then the transaction has to be aborted. If they are acceptable, then the MH can continue to execute the transaction further on the basis of the new results and with updated cache. Thus the transaction execution at MH is adjusted to the database state in the MSS. Here it should be noted that though the transaction tends to be long-running due to frequent and long disconnections, the actual transaction if executed on the fixed network would be much shorter.

Example 2 : Consider a case wherein a mobile user (perhaps travelling home by train) wishes to plan a pleasure cum business trip as shown in the Figure 4.2. He wishes to attend conferences in Delhi and Madras, and visit family and friends in London, Hyderabad, and Bangalore as shown in the Figure. The route and mode of transportation are also given in the figure. Hence, he has to book tickets for 9

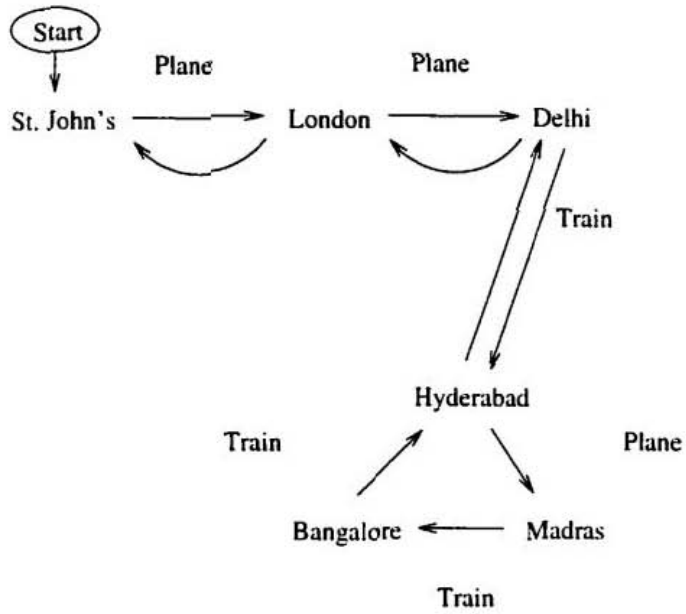


Figure 4.2: Pleasure cum business trip

legs. Furthermore, he needs to arrange for his hotel stay and car rental in London, Delhi and Madras. Due to the large number of airline companies that offer services across different cities, it is not possible to cache the entire information for all the legs. To begin with, the user caches flight information regarding the leg "St. John's $\longrightarrow$ London" and the information on hotels close to the Heathrow airport in London. The user selects the flight that best suits his travel plans and books a window seat that is available and also books a room in a nearby hotel in London. He tentatively executes the step of the transaction to book the ticket for this leg on the disconnected mobile host. He also provides an acceptance criteria to this step which states that - if window seat is unavailable, book any other seat. Another acceptance criteria could be such that if seat is not available on that particular flight 'A' - book a seat on a different flight 'B'.

After successfully executing the tentative step and specifying the acceptance criteria, he goes ahead to work on the next leg - "London $\longrightarrow$ Delhi". Since he has a conference to attend in Delhi, he might book a ticket with an airline which is more reliable though he may have better offers from other airlines. He might give an acceptance criteria to purchase a business class ticket on this flight if economy ticket is not available. Say at this point the MH reconnects to the MSS. The MSS validates all the previous steps of the transaction. At this point, the window seats of the particular flight in the previous leg may no longer be available. Thus the data read by the first step becomes stale. The MSS re-executes the step and checks for the acceptance criteria: if there are any seats available, books the same tentatively. The user proceeds with the transaction after caching the most up-to-date data. On the other hand, if both the acceptance criteria given in the first step are not satisfied, the step is aborted and the user is sent a diagnostic message.

After successfully validating or re-executing the steps, the user caches data and continues with booking on the new legs. At a future reconnection, the user validates step one, finds that there are no seats available on Flight A, so he re-executes this step again and tentatively books a seat on Flight B. The second step, "London $\longrightarrow$ Delhi" may also get invalidated as economy seats are no longer available. The step is re-executed and checked for acceptance criteria. If there are any Business class seats available, the transaction is allowed to proceed else aborted. Validation and re-execution proceeds till the transaction is complete and comes for final validation. □

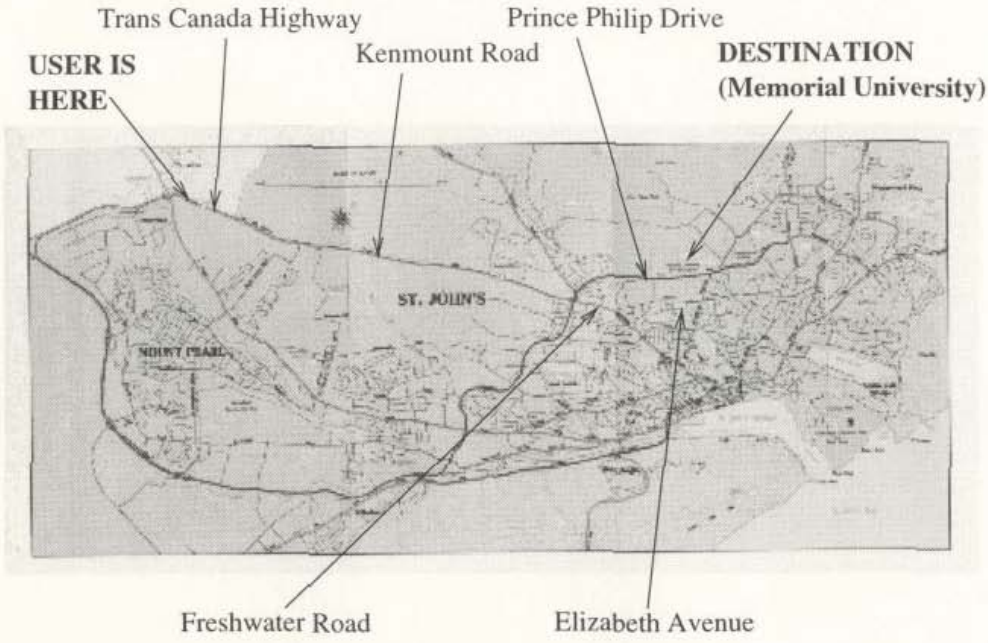
The idea of re-execution of transactions in the fixed network is appropriate in another context too. Memory restrictions may prohibit storing high-fidelity data on the MH. Then low-fidelity data can be cached and the computation performed at MH [56]. Later the transaction can be re-executed on high-fidelity data in the MSS. The coarse data on the MH can be viewed as enabling the user to get approximate results. Re-execution in MSS gives accurate results.

Example 3 : Let us imagine a scenario wherein a person travelling into a city needs directions to a particular destination. He connects to the MSS while approaching the city and caches the city map. Due to memory limitations of the MH, only a small map with coarse details may be cached. He may compute a possible route from that map, and then reconnect and submit the plan to MSS. The MSS may recompute the route on the basis of a very detailed map, which tells, for example, about one-way streets, number of lanes in highways at different times of the day and so on. The coarse data helps the user to come up with a coarse plan and start moving towards the destination. The re-execution of the transaction at MSS provides the exact route.

Let us consider a concrete example: a user wishes to attend a seminar in room EN -1054, Engineering Building, Memorial University of Newfoundland (MUN), St. John's.

1. While entering St. John's city from Trans Canada Highway (TCH), the user caches a coarse map of the city as shown in the Figure 4.3.

2. The user computes the route to MUN using the coarse data available as follows and submits it to the MSS.



Official Highway Map
Courtesy : Department of Tourism, Culture and Recreation, Newfoundland

Figure 4.3: Moving Towards Memorial University

TCH $\longrightarrow$ Kenmount Road $\longrightarrow$ Prince Philip Drive $\longrightarrow$ MUN.

3. (a) The MSS re-executes the route based on fine grained data available and *gives a more detailed route. It may also redirect the user on a slightly* different route after realizing an accident on Prince Philip Drive as follows.

   On TCH $\longrightarrow$ Exit to Kenmount Road $\longrightarrow$ Follow Kenmount Road which becomes Freshwater $\longrightarrow$ Take left at Elizabeth Avenue to reach MUN.

   (b) The MSS also gives the user a campus map as shown in Figure 4.4.

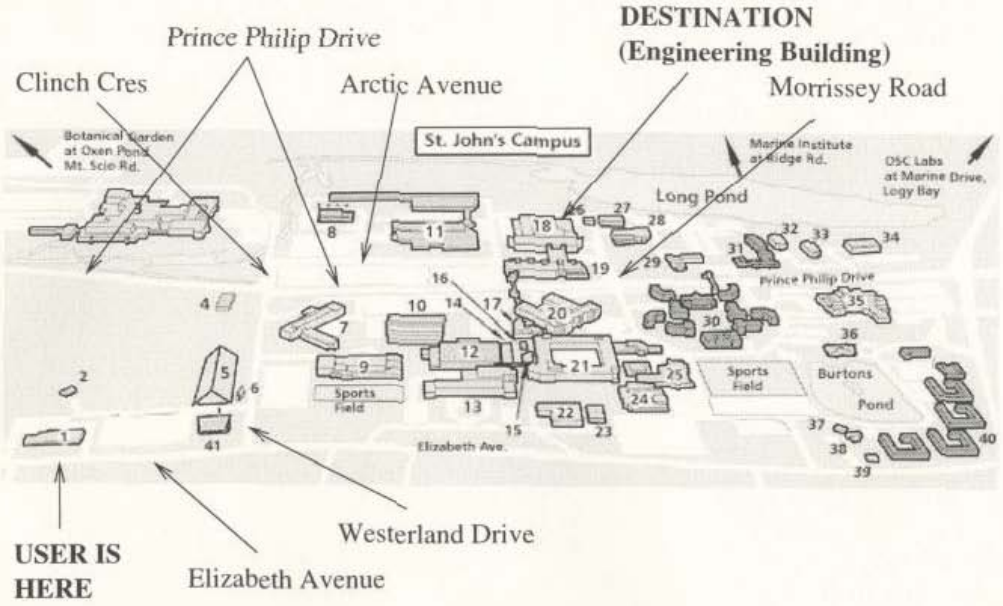


Figure 4.4: Memorial University Campus Map

4. The user uses this map to compute the route to Engineering building as follows and submits it to the MSS.

   From Elizabeth Avenue $\longrightarrow$ Left on Westerland $\longrightarrow$ Right on Prince Philip Drive $\longrightarrow$ Left on Morrissey Road $\longrightarrow$ Engineering Building.

5. (a) The MSS re-executes the route on fine grained data and finds that a left turn to Morrissey Road from Prince Philip Drive is not allowed. It recomputes a better feasible route as follows.

   From Elizabeth Avenue $\longrightarrow$ Take left on Westerland $\longrightarrow$ Follow Westerland which becomes Clinch Cres $\longrightarrow$ Take right on Arctic Avenue $\longrightarrow$ Engineering Building.

   (b) The MSS also gives the location of the parking lot and the map of the Engineering building as shown in the Figure 4.5

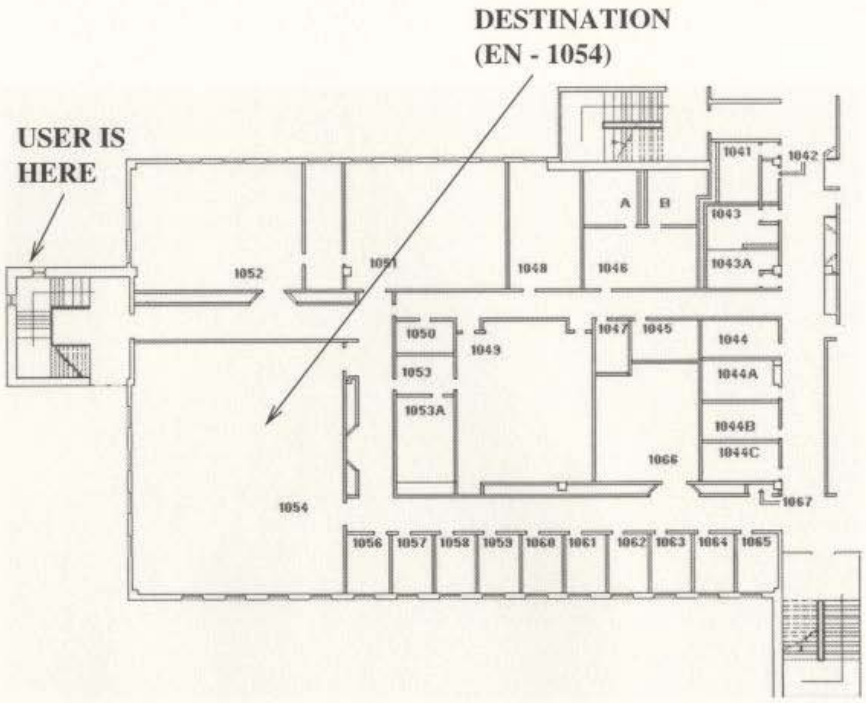


Figure 4.5: Engineering Building Map

6. The user can use this map to go to the room EN - 1054 as shown. □

Example 4 : Suppose ambulances are equipped with mobile computers to access a Medical Database of the people in a city. When an ambulance answers an emergency call, on finding a patient, the ambulance personnel may cache a brief medical history (coarse data) of the patient in the limited memory of the MH and use it to come up with an emergency treatment plan. This can be confirmed by re-execution on the MSS with the complete medical history containing, for example, major illnesses, allergies, etc. of the patient [12]. □

The new transaction execution model is presented in the next section.

4.2 New Transaction Model

As discussed earlier, mobile hosts can continue with execution of transactions in a disconnected mode. For this purpose, the mobile host caches a part of the database required for the execution of the ongoing transactions. For simplicity of exposition, we will assume the submission of a single interactive transaction to the MH in the following. We also assume, in this section, that the MH remains in the same cell during the execution of the transaction.

The *tentative* steps of the transactions executed at the MH produce *base* steps that are re-executed on the MSS. The re-execution adjusts the status of the ongoing transaction to be consistent with the database state of the MSS. The results of the re-execution must pass an *acceptance criteria* test. Otherwise, the transaction is aborted and the MH is informed of the same. The acceptance criteria gives the amount of divergence allowed on the results of the base transaction. Some sample acceptance criteria being:

- Quote value on shares purchased may diverge by + 0.1 % from the cached value.
- Quote value on shares sold may diverge by − 0.08 % from the cached value.
- Bank balance must not go negative.
- Though window seat is booked in a flight on the cached data of the MH, other seats may be booked if window seats are no longer available.

Also, the *validation need not be done with respect to the entire read set of the transaction, but with respect to only relevant data* which may be a subset of the read set. By doing this, we not only increase the possibility of successful validation (by

eliminating validations on irrelevant data read) but also speed up the validation process.

Example 5 : Let us consider a transaction to book an airline ticket from cities A to B. Initially the user may issue a step to find the most feasible routes from A to B - the query may be based on shortest distance, lowest price, etc. Let us assume that the query gives two results. A flight via cities X and Y with a stopover of one hour at each city, and another flight via city Z with a stopover of eight hours. The user may decide to pick the second route. By doing this, we can observe that the user's interest is now limited to data relevant to the flight via Z and therefore it suffices to validate the transaction with respect to this data alone, and not the entire data (including information about price, distance, timings of the first flight) in the step. □

Thus, some of the data values read in the early part of execution may be found to be irrelevant later on [49]. Then the validation needs to be done only with respect to the relevant data. We call this C-consistency [65]. We elaborate this concept first.

C-consistency: Let us assume that an interactive transaction consists of steps $s_1, s_2, \ldots, s_n$ as shown in Figure 4.6. Step s_1 reads data items a, b, c, d, e, and f and produces results p and q. The result p is based on data items a, b and d, and the result q on c, e and f. At this point, suppose the user decides that q is not useful (and so are c, e, and f) and decides to base the further execution only on p. Then there is no need to validate this transaction with respect to the values read for c, e, and f. In general, let D be the set of all data items read by the transaction, and C be the

subset of D that becomes eventually relevant for the computation. Then, validation needs to be based only on this subset C rather than the entire set D.

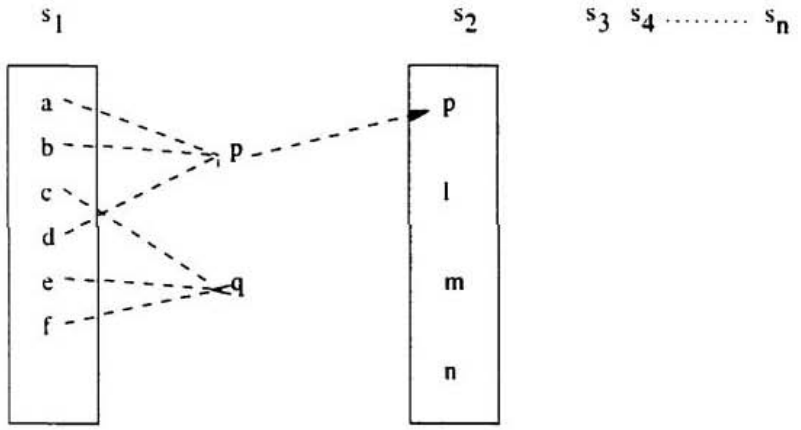


Figure 4.6: Data read in subsequent steps of an interactive transaction

Validation is done when the MH reconnects to the MSS, and whenever the user wishes to do so while in connected mode. The execution pattern could be like

$$s_1\ v_1\ s_2\ v_2\ s_3\ s_4\ s_5\ v_5\ \ldots\ s_{n-2}\ v_{n-2}\ s_{n-1}\ s_n\ v_n$$

where at point v_1, step s_1 is validated, at point v_2 both steps s_1 and s_2 are validated, and at point v_5 all steps up to s_5 are validated. Here, it should be noted that though the number of validations and re-executions are more, they may not be expensive since they are done on the fixed network. Also, an implementation may set a limit on the number of re-executions for each individual transaction.

Furthermore, the validation and re-execution of steps of the transaction is done with respect to the data present locally on the MSS. Each transaction is executed optimistically based on the local data available in the MSS and only when the transaction is complete does the global concurrency control mechanism come into picture.

The global concurrency control mechanism is then responsible to ensure that the data read is up-to-date and perform the necessary steps to ensure global serializability. Thus, other MSSs in the fixed network are not affected by this execution model until the transaction is submitted in full and comes for commitment.

4.2.1 Execution Model

For each MH, the MSS maintains a workspace with the following information.

- READSET - set of data items read by the steps executed so far by the MH.
- WRITESET - set of data items written by MH so far.
- MSS_STEPS - ordered sequence of steps of the transaction executed by MH so far.
- DOWNLOADSET - set of data items that were downloaded by the MH before the last disconnection.
- CHANGEDSET - set of data items that were downloaded by the MH and have been changed during disconnection by other transactions on the MSS.

In addition, it also keeps a *process image* containing the current values of the program counter, registers and variables of the program under execution.

The MH maintains the following sets along with the process image.

- MHREADSET - set of data items in the DOWNLOADSET read by the MH in disconnected mode since the last connection.
- MHWRITESET - set of data items written by MH in disconnected mode since the last connection.

- MH_STEPS - sequence of base steps of the tentative transaction executed by MH in disconnected mode since last connection.

4.2.1.1 Disconnection

1. (a) Just before disconnecting from the MSS, the MH caches the data required (DOWNLOADSET) to execute transactions in disconnected mode.

   (b) While being disconnected, the MH

   - executes the (tentative) steps of the transactions.
   - stores the data items that were read and written by the steps in the MHREADSET and MHWRITESET respectively.
   - creates the respective base steps with an acceptance criteria and stores them in MH_STEPS.
   - before every step, deletes any previous data items which become irrelevant as per the C-consistency concept from the MHREADSET, MHWRITESET and any steps affected from the MH_STEPS.
   - if the transaction is executed to completion in the disconnected mode, commits the transaction tentatively.

2. (a) When the MH disconnects from the MSS, the MSS stores the READSET, WRITESET, MSS_STEPS, DOWNLOADSET, and the current process image of the transaction in the workspace of the MH. The first three sets will be empty at the beginning of a transaction execution.

   (b) While the MH remains disconnected, the MSS keeps track of the data items which were cached by the MH at the time of disconnection, but are modified later, in the CHANGEDSET.

4.2.1.2 Reconnection

1. When the MH connects to the MSS:

   (a) The MH sends the MHREADSET, MHWRITESET, the process image, MH_STEPS (steps executed so far), along with any input parameters to the MSS and waits for the validation and, if necessary, re-execution of the base steps on the MSS.

   (b) If the transaction execution can continue, then it

   - accepts cache updates (correction of data items on cache) from the MSS,
   - the process image after the transaction steps were re-executed on the MSS, and
   - caches new data to continue execution of the transaction in disconnected mode.

2. When the MH connects to the MSS:

   (a) The MSS accepts the MHREADSET, MHWRITESET, the process image, MH_STEPS, any input parameters given by the user on the MH, and acceptance criteria, if any, and updates READSET, WRITESET and MSS_STEPS accordingly.

   (b) Then it checks for intersection between the data items present in CHANGEDSET and the MHREADSET.

   if intersection set **is not** ϕ
   some data read by the transaction has become inconsistent
   if re-execution allowed

```
            re-execute the steps affected as base steps.
            if within acceptance criteria
                if the transaction is executed to completion,
                    commit it.
                else (transaction is not complete)
                    move the new process image generated for the transaction
                    onto the MH and continue with the execution of the other
                    steps with new cached data on MH.
                endif
            else (acceptance criteria is not satisfied)
                abort the transaction.
                send a diagnostic message to the MH.
            endif
        else (re-execution of steps is not allowed)
            abort the transaction.
            send a diagnostic message to the MH.
        endif
    else
        transaction steps executed so far are valid.
        continue with execution of the transaction steps.
    endif.
```

When the transaction is either aborted or committed, the MSS empties the various sets maintained in the workspace.

4.3 Usability of the Model

In this chapter a new model for execution of transactions in a mobile environment is proposed. The underlying theme of the proposal is the *'tentative' execution of transaction in the MH and periodic confirmation of the execution in the MSS.* The confirmation may involve validation or re-execution of the transaction steps on the up-to-date data in the fixed network, using the (better) computing capabilities of the MSS. It has been argued that frequent re-executions help to adjust the course of com-

putation based on changes in the values of data (caused by the commitment of other transactions) and/or fine-grained computation on fine-grained data, or, in the worst case, abort the transaction much earlier than at the end of the execution. Our model is flexible enough to be applicable in a variety of mobile computing environments, and to address various transaction processing issues. Some aspects are illustrated in the following.

4.3.1 Adjusting the Computation

The concept of adjusting the computation, in accordance with the changes in the database state, is quite appealing for interactive transactions where future steps of the transaction are determined based on the results of the past steps. This is very useful in real time applications. For example, (i) a flight's course is re-adjusted based on the changing weather conditions ahead, (ii) the military routes are adjusted based on the enemy convoy's movements obtained from the radars, etc..

This is also very useful for collaborative executions. Several MHs in the same cell may be executing steps independently. All of them communicate the steps periodically to MSS. The MSS re-executes the steps of the different MHs, detects possible conflicts and suggests (global) adjustments to all the MHs. Thus, the MSS may act as a moderator of the collaboration. In this case the new values cached by the MH may reflect the (not-yet-committed, but) combined execution of the tentative steps of all the collaborating MHs.

4.3.2 Asymmetry in Communication

One of the characteristics of the mobile computing environment is the asymmetry in the cost of the communication between the MSS and the MH. Due to the resource

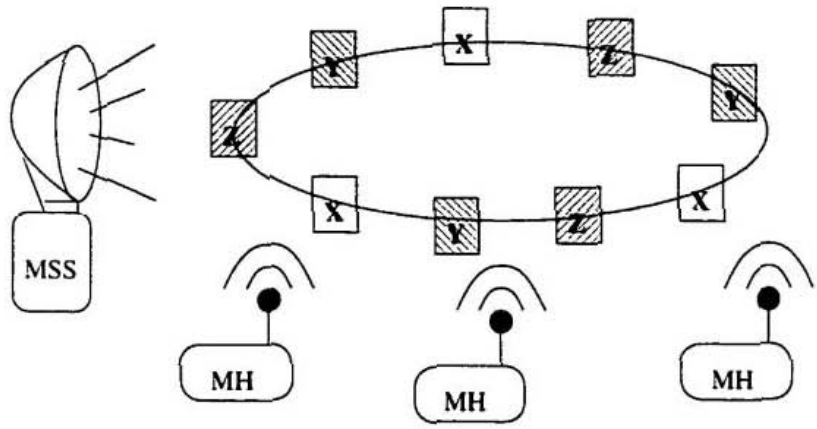


Figure 4.7: Broadcast Disk

constraints such as battery power, clients at mobile hosts usually have a weaker capability for transmitting messages than servers at fixed hosts or in other words, the available bandwidth from the MSS to the MH is significantly greater than the available bandwidth from the MH to the MSS. Thus, transmission of a message from a MH consumes more power than reception of a message of the same size at a MH.

The MSS on the other hand may have the broadcast capability in the wireless medium: that is, it can communicate any information simultaneously to *all* the MHs in its cell, at the same cost as it does for *one* MH. Hence, algorithms may be designed such that communication from MSS to MH is maximized and that from MH to MSS is minimized. This is achieved, for example, using the broadcast disk [1, 2, 27, 71] idea, that is, the relevant data and updates which are of interest to a set of MHs are repeatedly broadcast periodically by the MSS. In effect, the broadcast channel can be thought of as a spinning disk as shown in the Figure 4.7 from which MHs can retrieve data as it goes by.

Based on the data requirements of the MHs in the cell, the MSS can broadcast

some data pages more often than the others and hence keep that part of the database closer to the MHs. A hierarchy of data is formed by keeping some pages on high speed disks (broadcasting data pages more often) and keeping the others on low speed disks (broadcasting once in while). A MH will have to wait longer to access data on the low speed disks. Some example broadcast programs are shown in the Figure 4.8. Compared to flat broadcast, the subsequent broadcasts of data item X are potentially clustered in the skewed model. The multidisk broadcast gives the feeling that the data item X is stored on a disk that is spinning twice as faster as the disk containing Y and Z.

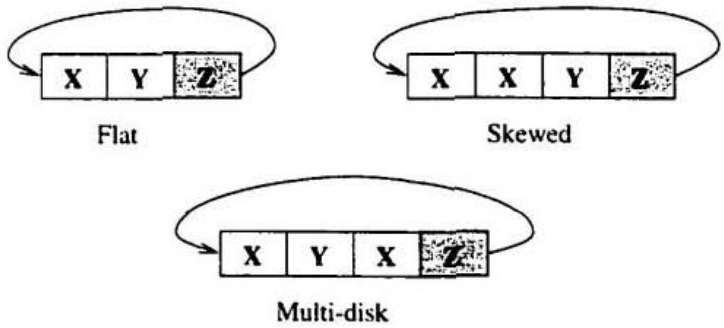


Figure 4.8: Example Broadcast Programs

The proposed model can be modified to suit the environment as follows. Caching which is inherent in our model reduces the contention on the narrow bandwidth wireless channels. Though the data that is cached by several MHs is placed on low speed disks in the broadcast disks, it can be moved onto high speed disks once it becomes stale due to updates by other MHs. Thus, periodically the MH may refresh the cache and re-execute the steps locally and adjust the future course of the computation. On reconnection, the MH need not re-execute the steps again, but rather cache new data and proceed with the computation.

4.3.3 Weak Connection

Another characteristic of the mobile computing environment is the weak connection. Weak connection may constrain the information transfer between MH and MSS. Amounts of information to be transferred can be reduced in two ways. Suppose the steps s_1 and s_2 have been executed at MH at the time of a (weak) reconnection. (i) The MH-state at s_1 can be sent to MSS, instead of the MH-state at s_2: the sizes of the sets might be smaller, and also the response from validation/re-execution of s_1 at MSS might come a little sooner. Then, based on the response, the MH itself may re-execute s_2. (ii) Further reduction can be achieved by only sending the MH-state at s_1 to MSS in one connection, and receiving the response from MSS in another connection, that is by de-coupling steps 4.2.1.2.1(a) and 4.2.1.2.1(b). In the mean time, the MH may continue with the next step, say s_3, and, after the second connection, may re-execute both s_2 and s_3, if needed.

4.3.4 Releasing Intermediate Results

In the model a sequence of operations have been grouped into steps, and a sequence of steps as transaction. One may assume an open nested transaction model as in [38], and treat steps as subtransactions. Then, as soon as validation/re-execution of a step is done at MSS, that step can be committed and its effects released. In section 4.2, again for simplicity, the execution of a single transaction on MH was considered in the description of the method. A disconnection period may be so long that the transaction currently executed is completed, and some new transactions are also executed. The completed transaction cannot be committed since this requires communication with the MSS. Instead a different kind of action [25], a "tentative commit" is used

which records an intention to commit and allows the mobile host to start up the next transaction. Having tentative commits leads to "dependent commits" [40]: transaction T2 depends on T1 if it uses objects modified by T1 because if T1 ultimately aborts, so must T2. At the time of reconnection, the MSS would re-execute all of them and try to commit them in the same tentative commit order as in the MH [24]; and the last not-yet-finished transaction would be re-executed and possible adjustments communicated to the MH, as per the proposed algorithm.

4.3.5 Mobility

The mobile transaction, compared to traditional transactions, may not begin and terminate on the same MSS. That is, the MH executing the mobile transaction may move (hop) across several cells, connect to different MSSs, partially execute the transaction at each site, and finally terminate at a different site. As depicted in Figure 4.9, the mobile transaction may:

- be executed in its entirety in the same cell (on the same MSS) without any hops.
- begin and end on the same MSS, after moving (hopping) across several cells.
- begin at some MSS, hop across several cells, and finally terminate on a different MSS.

Our algorithm can be modified in the following ways to account for such mobility:

4.3.5.1 Re-execution

Since the transaction execution on the MSS is optimistic in nature, when the MH moves to a new cell, the steps of the transaction executed so far can be re-executed on

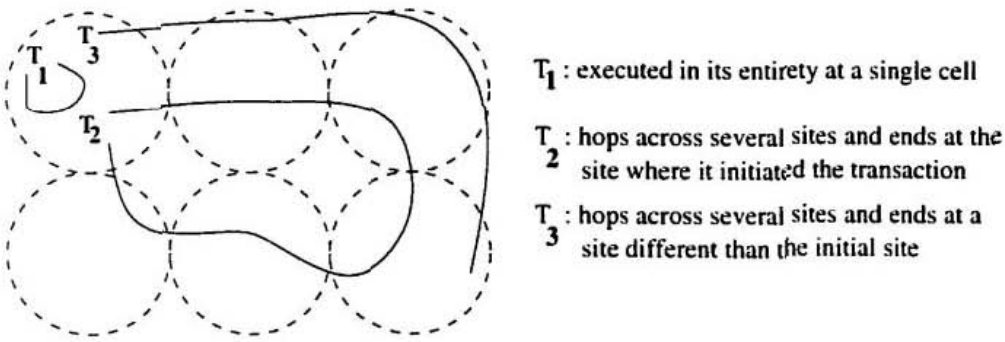


Figure 4.9: Movement of a MH across cells while executing a transaction

the new MSS and the transaction execution continued. That is, minimal information like the steps executed, input parameters submitted by the user, and any acceptance criteria given are transferred from the previous MSS to the new MSS during the hand-off process and the transaction re-executed. On successful transaction re-execution and creation of workspace, the workspace on the source MSS is purged and complete control of transaction execution given to the destination MSS.

4.3.5.2 Home-sites

To capture the mobility aspect of the MH in the transaction model proposed in this thesis, the architecture of the mobile database system may be extended to include *home-sites* [12]. Each MH in the system is affiliated with a home-site which provides a central repository for all the MH's transaction processing activity. Hardware coded identification numbers are employed to uniquely identify all the MHs and MSSs. A mapping table (address directory - AD) which contains all the MHs and MSSs along with their associations is maintained at each MSS as shown in Figure 4.10. There is a many-to-one mapping between the MHs and MSSs as a single MSS can serve as an home-site to several MHs. Also, each MH can have only one MSS as its home-site.

Thus when MH_1 connects to MSS_5 as shown in the Figure 4.10, MSS_5 identifies the mobile host as a nomadic MH (or roaming MH). It looks up in the address directory (AD) and locates the home-site MSS_1 for the MH_1 and informs the current location of MH_1 to MSS_1.

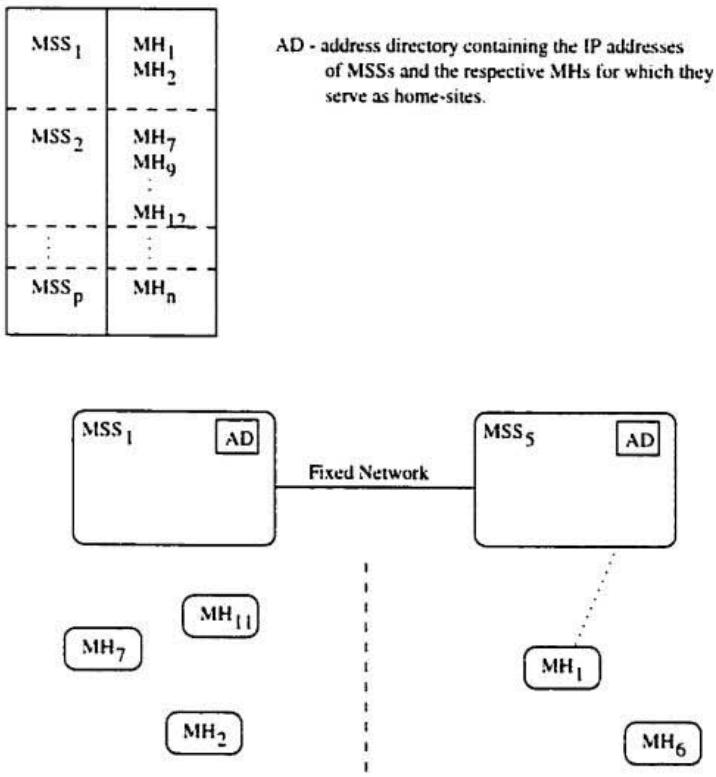


Figure 4.10: Home-sites for MHs

The home-sites take care of the mobility aspect as follows:

1. Delegate the responsibility of execution to the home-site.

   This way, the primary function of the MSS will be providing an interface between wired and wireless networks, and not any application related computation. Then, irrespective of the location of the MH, the MSS communicating

with the MH will forward all information from the MH to its home-site and vice versa. An optimization would be to assign and make use of home-site only for transactions during whose execution the MH moves from one cell to another. Thus, if the transaction executes in its entirety in a single cell, the current MSS holds the responsibility of the execution.

2. Treat the home-site as location coordinator for the MH.

   Whenever an active (connected) MH moves across the cell, a *hand-off* process occurs. The hand-off process involves informing the new destination MSS_k where the MH is coming from and transferring the workspace of the transaction to the new MSS_k. The workspace on source MSS_j is deleted or purged on successful hand-off. The transaction execution can continue on the new MSS.

   On the other hand, if the MH disconnects from the MSS, the workspace is moved to the home-site of the MH. On reconnection at any MSS, the home-site is contacted and the workspace recovered at the new MSS and transaction execution continued. An optimization would be to just send the location of the last MSS which has the workspace to the home-site. On MH's reconnection at any MSS in the fixed network, the home-site prompts the last MSS to forward the workspace to the new destination MSS.

3. Home-site can act as a coordinator.

   The MH can execute different steps of the transaction at different MSSs as it moves. The home-site is responsible to check with each site at specified intervals and re-execute any steps if necessary.

4.3.6 Recovery

Mobile hosts are more susceptible than fixed hosts to both communication and station failures. They are also more prone to theft, loss, or accidental destruction. Therefore, it is recommended [33, 47] that, to ensure durability, part of the computation that is performed on the MH be recorded in the fixed network to achieve persistency. The proposed execution model facilitates this naturally. At every point of reconnection, after the transaction steps are successfully validated/re-executed, the process image of the ongoing transaction along with log information is stored on the MSS. That is, the process image is backed up periodically at the stable storage - in our case, of that of the MSS. Upon a failure, the process rolls back to the most recent *back-up* process image available on the MSS and continues execution. Though the MH's computation in the last disconnected period is lost the transaction can still recover from the point of its last reconnection. The backups of process image on the MSS occurs at every reconnection in our model. We may as well back-up the process image at some regular intervals instead of at every reconnection. These intervals could be based on the (a) number of steps executed, (b) time elapsed, and (c) user's choice. The mobility of the MH on the other hand will require moving the process image along with it. When a MH moves across to a different cell, the process image stored at the previous MSS is moved onto the new MSS during the hand-off process. Thus if the MH fails, it may retrieve the process image from the new MSS ensuring quick recovery at the cost of *delay* observed in hand-off due to transfer of process image. This strategy is well suited for applications where long service disruptions are not tolerable and where the failure rate of the MHs is high. The concept of home-sites for MHs may be employed to deal with the situation where in a MH fails, moves across

to a different cell and tries to recover. In this case, the home-site can keep track of the last MSS that has saved the process image and on MH's reconnection at any part of the network can request this MSS to forward the process image to the respective MSS which is currently serving the MH.

4.3.7 Pessimistic mode of Operation

The execution model incorporates optimistic approach. If desired, the execution can be shifted to pessimistic mode at any stage. For example, after validation/re-execution of some steps, it may be determined that the rest of the computation at MH should be confirmed as such at MSS also. Then the relevant data items can be 'locked' at MSS, prohibiting access by other transactions, until the MH executes the remaining steps. The commitment at the MH would then automatically imply commitment at MSS at the next connection. Note that this way the number of items that are locked and the duration of the locks are minimized. This is explored in the next chapter.

Chapter 5

INTEGRATED APPROACH

5.1 Introduction

In the previous chapter, we proposed *to extend the re-execution idea to various intermediate stages of execution of a transaction* at MH. At every stage (at reconnection points), all the steps which have been executed at MH thus far are validated, and, if the validation fails, re-executed at MSS, and the results are checked against the acceptance criteria. If the new results are acceptable, then the MH can continue to execute the transaction further on the basis of the new results and with updated cache, else the transaction is aborted. Thus the transaction execution at MH is *adjusted* to the database state in the MSS. We also argued that validation need not be done with respect to the entire read set of the transaction, but with respect to only relevant data which may be a subset of the read set. (Some of the data read may have become irrelevant in the course of the computation due to c-consistency.) This further increases the possibility of successful validation.

Due to the inherent nature of the optimistic approach, despite frequent validations and re-executions along the way, transactions could still be aborted when they come for commitment. That is, the MH's computation is not guaranteed until the

MSS commits it at a later stage. In this chapter, we explore some ways of guaranteeing against invalidation, and thus enhancing the utility of MH's computation. We illustrate this with an example.

Example 1 : Let us consider a transaction to book airline tickets from cities A to F via cities B, C, D, E. This process includes 5 steps, booking each leg (A-B, B-C, C-D, D-E, E-F) separately (due to memory restrictions). While executing step 2, the user observes that the number of seats in the leg B to C is not many. The user may proceed with the other steps and when he tries to commit the transaction, he may find that there are no seats left on leg 2 thus forcing to abort the entire transaction. If it were possible to 'reserve' the seats in that leg, 'at least for a short time, within which the remaining computation *might* finish', the transaction might not have been aborted at the end. □

In this chapter, we aim to provide such guarantee. It is obvious that providing any guarantee involves some kind of 'locking' or pessimistic approach: and this will restrict or delay other (conflicting) transactions' executions. These effects are significant in mobile environments since MHs (holding locks) may be disconnected from the fixed network for long, unpredictable, durations. We accommodate this problem in two ways: (i) varying degrees of pessimism are defined (allowed) and (ii) a timeout period is associated with each pessimistic access. Several factors may determine both the degree of pessimism and the duration of the timeouts. The timeout period is the estimated time interval within which the transaction is expected to commit. If the commitment does not occur (for example, the MH does not reconnect within that period), then the pessimistic access is switched to optimistic one. The degrees of pessimism are those introduced in [66]; here they are tailored to the mobile environ-

ment. They reflect conflict-level integration of optimistic and pessimistic concurrency control.

The features of the integrated concurrency control method are described in section 2, and the new model for transaction execution in mobile environments in section 3. Section 4 gives the correctness proof.

5.2 Integrated Concurrency Control Method

Two transactions are said to be *conflicting* if the write set of one intersects with either the read set (RW-conflict), or the write set (WW-conflict) of the other. The main concern of the concurrency control mechanism is to correctly process conflicting transactions. The *capability structure* described below not only achieves this but also provides the MH's user with the flexibility of switching the transaction execution between optimistic and pessimistic modes at any stage.

A transaction that wants to read (write) a data item must first obtain a *read (write) capability* (permit to access) for that data item. The capabilities may have *priority* or *no_priority* with respect to conflicts. Priorities provide pessimistic access to data items and no_priorities optimistic access. Therefore, we have (i) two types of read capabilities, P_RW and NP_RW indicating priority or no_priority with respect to RW-conflicts, and (ii) four types of write capabilities, (P_RW P_WW), (P_RW NP_WW), (NP_RW P_WW), (NP_RW NP_WW), for the two options for the two types of conflicts.

Priority with respect to a conflict facilitates automatic validation with respect to that conflict in the validation phase. That is, a P_RW read capability for data item x guarantees that at the time of validation no other transaction would have modified

the value of x. On the other hand, a NP_RW read capability for data item x implies no guarantee and hence waiting until concurrent transactions with P_RW write capability on x (i) commit, in which case the read capability is revoked, or (ii) abort, in which case the read capability can be validated successfully. A (P_RW NP_WW) write capability for x implies guarantee and hence no-waiting against concurrent transactions reading x, but no guarantee and therefore waiting until concurrent transactions writing x finish commitment (or abort). A (NP_RW P_WW) write capability for x implies waiting until all transactions with P_RW read capability for x finish but no waiting against concurrent transactions writing x. A (P_RW P_WW) write capability for x implies no waiting against transactions reading or writing x. On the other hand, a (NP_RW NP_WW) write capability implies no guarantee and hence waiting until concurrent transactions holding P_RW read capabilities or P_WW write capabilities finish commitment (or abort). This is analogous to an optimistic write operation.

The capabilities are granted according to the compatibility matrix given in the Figure 5.1. The conditional 'Y', 'Y^1' represents concurrent write capabilities with priorities allowed by the application. The number of such capabilities may be decided by MSS based on the maximum number of transactions holding the writes on x, the user, or the time at which the request is made. For example, in a stock application, each user may be allowed to buy a limited number of shares, for instance, a maximum of 10 shares of a company. If only 100 shares are available, then the MSS can allow upto 10 stockbrokers to hold write capabilities in P_WW mode concurrently. When concurrent write capabilities with priorities are not allowed, 'Y^1' is replaced by 'N'. Allowing concurrent writes is analogous to the escrow transactional model [45] which was designed specifically to improve concurrent access to *aggregate items* by exploiting

object structure, state-based commutativity, and integrity constraints. The escrow model exploits the fact that aggregate items are numerical values which represent a quantity of interchangeable items (example, number of shares of a company available or dollars in an account). Thus the quantity can be divided among a number of mobile hosts based on the data requirements [68]. The compatibility matrix presented allows this feature naturally.

		READS		WRITES			
		P_RW	NP_RW	P_RW P_WW	NP_RW P_WW	P_RW NP_WW	NP_RW NP_WW
READS	P_RW	Y	Y	N	Y	N	Y
	NP_RW	Y	Y	Y	Y	Y	Y
WRITES	P_RW P_WW	N	Y	Y^1	Y^1	Y	Y
	NP_RW P_WW	Y	Y	Y^1	Y^1	Y	Y
	P_RW NP_WW	N	Y	Y	Y	Y	Y
	NP_RW NP_WW	Y	Y	Y	Y	Y	Y

Figure 5.1: Compatibility Matrix

We note that an NP_RW read capability for x can be issued to T_i irrespective of any other transaction T_j holding any capability for x. That is, data item x can be allowed access in both modes P_RW and NP_RW by different transactions at the same time. The notation in Table 5.1 is used to represent the respective capabilities.

The transaction processing in an MH can be considered to consist of five (not necessarily distinct) phases:

1. **Request and acquisition of capabilities:** A transaction must acquire the

Capability	Notation
read in P_RW mode	r^p
read in NP_RW mode	r^n
write in (P_RW P_WW) mode	w^{pp}
write in (P_RW NP_WW) mode	w^{pn}
write in (NP_RW P_WW) mode	w^{np}
write in (NP_RW NP_WW) mode	w^{nn}

Table 5.1: Notation for respective read and write capabilities

respective read or write capabilities before performing the operations. If a request for a capability cannot be granted right away, respective optimistic read and write capabilities are granted as default and the transaction allowed to proceed in an optimistic fashion. The capabilities are acquired based on the rules in the compatibility matrix. (We note from the compatibility matrix that the optimistic capabilities r^n and w^{nn} can always be granted.) Some of the capabilities may get revoked due to timeout periods or committing of other transactions. These capabilities are re-obtained and steps re-executed.

2. **Execution:** Each MH is provided with a *private workspace* on the MSS. All the MH's tentative transaction writes are done first in this private work space, and only after successful final validation are they done in the database itself. The private workspace also holds relevant information about the transaction in execution. What information is stored will be discussed in detail in the next section.

3. **Final validation:** The final validation is different from the intermediate validations during the transaction execution in our model. Final validation can be thought of as conversion of the capabilities into pessimistic capabilities, that

is, (read and write) locks. This is the "locking" phase. Priority mode implies automatic conversion. No_priority mode requires waiting until all transactions holding conflicting pessimistic mode capabilities commit, abort, or release the capabilities. Of course, NP_RW read capability cannot be converted into read lock if another transaction with P_RW write capability commits.

4. **Write phase:** Writes are transferred from the private workspace to the database after successful final validation.

5. **Release of capabilities:** All the capabilities held by the transaction are released simultaneously after the write phase or the abortion of the transaction. This is the "unlocking" phase.

We assume in this thesis, that the last three phases, that is, the final validation, the write phase, and the release of capabilities, are done in a critical section. We note that two phase locking policy is followed here.

5.3 Transaction Execution Model

Mobile hosts can continue with execution of transactions in a disconnected mode. For this purpose, the MH caches a part of the database required for the execution of the ongoing transactions. For simplicity of exposition, we assume the submission of a single interactive transaction to the MH in the following. We also assume, in this chapter, that the MH remains in the same cell during the execution of the transaction.

An example sequence of steps of an interactive transaction could be as shown in Figure 5.3. The arrows indicate the dependencies between the steps. While step s_0

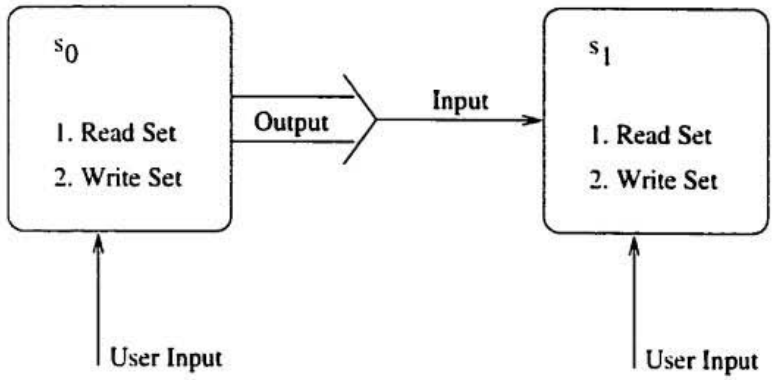


Figure 5.2: Dependency between steps

triggers two different and independent steps s_1 and s_4. s_{11} depends on the output of s_{10}, and s_9 requires input from both steps s_6 and s_8. On the other hand, s_3 and s_{10} are independent steps.

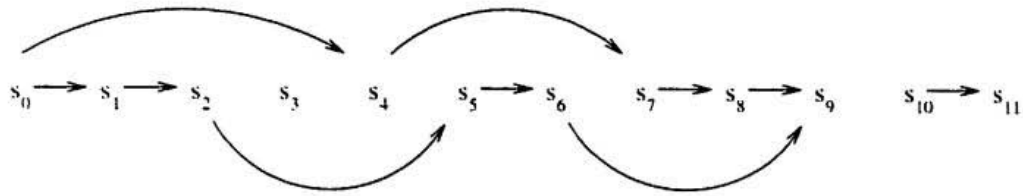


Figure 5.3: An example sequence of steps

By default, the transactions are granted no_priority capabilities in MH. Specific pessimistic capabilities are requested during reconnection with MSS. The MSS issues the capabilities to MH's transactions based on the capability structure described above. Some capabilities may be granted right away. Some others may be issued only in subsequent reconnections. Timeout periods are introduced for capabilities with priorities to safeguard against the 'number of disconnections' and 'periods of disconnections'. Once the timeout period expires, the MSS can revoke the capability. Timeout periods could be based on the application, the time at which the priorities

are made, or some tariffs (user may be charged according to the timeout period). The MSS keeps track of the data items read (written), capabilities held, transaction steps executed so far, etc., both for recovery purposes and management of capabilities while MH is disconnected.

The *Capability Granting Manager* (CGM) on the MSS is responsible for granting or revoking of capabilities to transactions submitted by different MHs. The capability requests are maintained in a FIFO queue and capabilities granted in that order. If a priority capability is requested and it cannot be granted, a no_priority capability is granted and the following requests that do not conflict with the waiting request are serviced. A status structure for each data item x is maintained as shown below which will give the number of different read/write capabilities held by the different transactions on x. This is used by the CGM to either grant or delay the capability requests arriving as per the capability granting rules. With each data item we also associate a CAP_SET which contains all transactions currently holding the optimistic read capabilities on x.

struct STATUSX {
 NR_P int: *No. of transactions holding reads in* r^p *mode.*
 NR_NP int: *No. of transactions holding reads in* r^n *mode.*
 NW_P int: *No. of transactions holding writes in* w^{pp} *mode.*
 NW_P1 int: *No. of transactions holding writes in* w^{pn} *mode.*
 NW_P2 int: *No. of transactions holding writes in* w^{np} *mode.*
 NW_NP int: *No. of transactions holding writes in* w^{nn} *mode.*
}

Granting of Read Capabilities to transaction T_i.
Procedure *Process_Read*$(R_i(x), C_i)$

```
begin
   if (C_i = r^p)
      if (NW_P = 0 and NW_P1 = 0)
         issue r^p capability to T_i;
         associate a timeout value with the capability
         (in case r^n is already held, then upgrade it to r^p)
         STATUSX.NR_P := STATUSX.NR_P + 1;
      else
         delay the request and issue capability when condition satisfied:
         issue no_priority read capability (r^n):
         STATUSX.NR_NP := STATUSX.NR_NP + 1;
         CAP_SETX := CAP_SETX ∪ T_i;
      endif
   else
      issue r^n capability to T_i;
      STATUSX.NR_NP := STATUSX.NR_NP + 1;
      CAP_SETX := CAP_SETX ∪ T_i;
   endif
end.
```

Granting of Write Capabilities to transaction T_i.
Procedure *Process_Write*$(W_i(x), C_i)$

```
1. if (C_i = w^pp)

   begin
      if (NR_P = 0 and (NW_P + NW_P2) < 'LIMIT')
         ('LIMIT' is the maximum number of concurrent writes allowed):
         issue w^pp to T_i;
         associate a timeout value with the capability
         STATUSX.NW_P := STATUSX.NW_P + 1;
      else
         delay the request and issue capability when condition satisfied:
      endif
   end.
```

2. **if** ($C_i = w^{pn}$)

begin
if (NR_P = 0)
issue w^{pn} to T_i;
associate a timeout value with the capability
STATUSX.NW_P1 := STATUSX.NW_P1 + 1;
else
delay the request and issue capability when condition satisfied;
endif
end.

3. **if** ($C_i = w^{np}$)

begin
if ((NW_P + NW_P2) < 'LIMIT')
issue w^{np} to T_i;
associate a timeout value with the capability
STATUSX.NW_P2 := STATUSX.NW_P2 + 1;
else
delay the request and issue capability when condition satisfied;
endif
end.

4. **if** ($C_i = w^{nn}$)

begin
issue w^{nn} to T_i;
STATUSX.NW_NP := STATUSX.NW_NP + 1;
end.

Whenever a capability is revoked, the respective values in the structure are decremented accordingly. For granting priority capabilities, the MSS must contact all other nodes in the fixed network to ensure that no other transaction is holding conflicting capability.

In the following tables we will visualize the kind of capabilities that can be held on a data item x by different transactions at any instance following the rules of the

compatibility matrix. We also assume the value of LIMIT to be '1', that is, concurrent write capabilities are not allowed. A '$\Diamond$' represents a capability already held by some transaction. A '$\surd$' represent a capability granted to a transaction after '$\Diamond$' was already issued. An exclusive write lock is assumed to allow read lock also. Thus, if w^{pp} is held by a transaction T_i, then T_i is also allowed to read in r^p mode. Also, with w^{pn}, if no other transaction holds a read r^p mode then the transaction can be granted the read in r^p mode.

1. T_i currently holds a read capability in P_RW mode on x :

r^p	r^n	w^{pp}	w^{pn}	w^{np}	w^{nn}
$\Diamond$	many			1	many

Table 5.2: T_i holds read in P_RW mode. $\Diamond$ = 1 or many.

Since T_i is already holding read capability in P_RW mode on data item x, no other transaction is allowed to obtain writes in P_RW mode. However, NP_RW reads and NP_RW write capabilities are allowed. Also, the value of '$\Diamond$' can be 'one or many' as several transactions can hold priority read capabilities simultaneously.

2. T_i is currently holding a read capability in NP_RW mode on x:

As shown in the first row of Table 5.3, '$\Diamond$' ('one or more' read capabilities in NP_RW mode) is currently held by T_i and T_j is granted the next '$\surd$' (read capability in P_RW mode). The other possible capabilities that could be granted on data item x in this scenario are (i) a write capability in (NP_RW P_WW) mode and (ii) several write capabilities in (NP_RW NP_WW) mode.

r^p	r^n	w^{pp}	w^{pn}	w^{np}	w^{nn}
√	◇			1	many
	◇	√	many		many
	◇		√	1	many
	◇	1	√		many
	◇		many	√	many
many	◇			√	many

Table 5.3: T_i holds read in NP_RW mode. ◇ = 1 or many.

The other rows and the following tables are to be interpreted similarly based on the compatibility matrix.

r^p	r^n	w^{pp}	w^{pn}	w^{np}	w^{nn}
	many	◇	many		many

Table 5.4: T_i holds write in (P_RW P_WW) mode. ◇ = 1.

r^p	r^n	w^{pp}	w^{pn}	w^{np}	w^{nn}
	many	√	◇		many
	many		◇	√	many

Table 5.5: T_i holds write in (P_RW NP_WW) mode. ◇ = 1 or many.

r^p	r^n	w^{pp}	w^{pn}	w^{np}	w^{nn}
√	many			◇	many
	many		√	◇	many

Table 5.6: T_i holds write in (NP_RW P_WW) mode. ◇ = 1.

r^p	r^n	w^{pp}	w^{pn}	w^{np}	w^{nn}
√	many			1	◇
	many	√	many		◇
	many		√	1	◇
	many	1	√		◇
	many		many	√	◇
many	many			√	◇

Table 5.7: T_i holds write in (NP_RW NP_WW) mode. ◇ = 1 or many.

Logical Clocks

The MSS maintains a logical local clock for each data item to aid in validating data read by an ongoing transaction. The clock gives the logical time when the data was created or updated. The logical clock CT^x maintained for each data item x is a positive integer 'i'. When the data item x is created or updated, CT^x is incremented by '1'.

Let us assume that a transaction is using data item x with clock value CT^x_{t1} at time $t1$. When it comes for validation later at some time $t2$ it checks with the current clock value CT^x_{t2} available on the MSS for the data item x.

procedure validate(CT^x_{t1}, CT^x_{t2})

begin
    valid := true;
    if $CT^x_{t1} < CT^x_{t2}$ **then** valid := false;
    if valid **then**
        value of the data item has not changed;
    else a new version for data item has been created;
end;

Data Structures

The MSS maintains a workspace with the following information for each MH:

- MSS_REQ_SET - Contains the set of capabilities requested by T_i, but not yet granted.
- MSS_GR_SET - Contains the set of requested capabilities that have been granted already.
- MSS_REV_SET - Set of capabilities that were granted, but later revoked due to expiration of timeout periods or commitment of other transactions.
- MSS_STEPS - Partially ordered sequence of steps of the transaction executed by MH so far.
- MSS_AFF_STEPS - Steps affected due to revoking of capabilities on data items used by the transaction.

Whenever a capability is granted for a request present in MSS_REQ_SET, the data item is deleted from the MSS_REQ_SET and placed in the MSS_GR_SET. When the capability is revoked, it is removed from the MSS_GR_SET and placed in the MSS_REV_SET. The MSS_STEPS affected thereby are moved to MSS_AFF_STEPS to be later re-executed.

In addition to this, the MSS keeps the process image containing the current values of the program counter, registers and variables of the program under execution in the MH.

In addition to the process image the MH maintains the following information:

- MH_OLD_REQ_SET - Capabilities held by the transaction on the MH before the last reconnection.
- MH_NEW_REQ_SET - Capabilities requested so far in the current disconnected period.
- MH_DEL_SET - Capabilities not required by the transaction anymore due to c-consistency.
- MH_GR_SET - Capabilities that have been already granted on the MSS.
- MH_REQ_CACHE - Data items the MH tried to access in disconnected mode, but were not present in the current cache of the MH.
- MH_OLD_STEPS - Sequence of all partially ordered base steps executed by MH in disconnected mode before last reconnection along with any dependencies among them.
- MH_NEW_STEPS - Sequence of partially ordered base steps executed by MH in disconnected mode since last reconnection along with any dependencies among them.
- MH_DEL_STEPS - Base steps executed by MH in disconnected mode before last reconnection no longer needed due to c-consistency.

5.3.1 Disconnection

1. (a) Just before disconnecting from the MSS, the MH caches the MH_REQ_CACHE and any other data required to continue execution of

the transaction in disconnected mode. It also tries to obtain the required capabilities for these cached data items if already known. The granted capabilities on data items for the transaction are maintained both on the MH and the MSS. The following updates are made :

MH_GR_SET = MSS_GR_SET;

(b) While disconnected, the MH

- executes the (*tentative*) steps of the transaction. If the data item read or written is in the cache (*cache hit*) and requested capability is not present for the data item in MH_GR_SET, logs the new capability request in MH_NEW_REQ_SET. Continue with execution of transaction steps.

  If the data item to be read is not present (*cache miss*), the request for the data item along with any capability requests is logged in the MH_REQ_CACHE for caching at the point of next reconnection. Proceed with other steps of the transaction if possible or wait for the next reconnection.

  If data item to be written is not present, create the data item using a template and write the value in it. Log in data items and any capability requests in MH_NEW_REQ_SET. If template for the data item is not available to create the data item, the writes are delayed until reconnection time.

- creates the respective partially ordered base steps with an acceptance criteria for all steps executed so far by transaction T_i in the discon-

nected mode and stores them in the MH_NEW_STEPS.

- before every step, deletes any previous data items which become irrelevant as per the c-consistency concept from MH_OLD_REQ_SET, MH_NEW_REQ_SET, and MH_GR_SET. The MH stores the data items that had been deleted from the MH_OLD_REQ_SET in the MH_DEL_SET as these have to be updated on the MSS. Similarly delete all steps no longer required from the MH_OLD_STEPS, MH_NEW_STEPS and store those steps deleted from MH_OLD_STEPS in MH_DEL_STEPS to be updated on the MSS.
- if transaction is executed to completion in disconnected mode, commit the transaction tentatively.

2. (a) When the MH disconnects from the MSS, the MSS stores the MSS_REQ_SET, MSS_GR_SET, MSS_REV_SET, MSS_STEPS, MSS_AFF_STEPS, and the current process image of the transaction in the private workspace of the MH. The first five sets will be empty at the beginning of the transaction execution.

   (b) While the MH remains disconnected, the MSS

   - tries to obtain the read and write capabilities which were not granted at the time of last reconnection indicated by the non-empty FIFO queue or the MSS_REQ_SET. This might be due to some other transaction T_j which is holding capabilities with priorities over that data item.
   - places the revoked data items in the MSS_REV_SET and deletes the

same from the MSS_GR_SET. The steps affected are moved from MSS_STEPS to MSS_AFF_STEPS. It then waits for MH to reconnect and take the necessary action.

5.3.2 Reconnection

1. When the MH connects to the MSS:

   (a) The MH sends the MH_NEW_REQ_SET, MH_REQ_CACHE, MH_DEL_SET, MH_NEW_STEPS, MH_DEL_STEPS, along with the process image and input parameters given by the user on MH, and acceptance criteria, if any, and waits for validation or re-execution of the base steps on the MSS. (The validation at this point is not to convert capabilities to locks. This conversion is done only at the end of the transaction execution when it comes for commitment.)

   (b) If transaction T_i is successfully validated or re-executed, the MH

   - accepts the cache updates from the MSS.
   - caches new data (and MH_REQ_CACHE, if present) to continue execution of T_i in disconnected mode.
   - may obtain capabilities for new data items being cached.

2. When the MH connects to the MSS:

   (a) The MSS accepts the MH_NEW_REQ_SET, MH_REQ_CACHE, MH_DEL_SET, MH_NEW_STEPS, MH_DEL_STEPS along with the process image, acceptance criteria, and any input parameters given by the user on the MH. It then updates the following sets:

MSS_REV_SET = (MSS_REV_SET − MH_DEL_SET);

MSS_REQ_SET = (MSS_REQ_SET − MH_DEL_SET);

MSS_GR_SET = (MSS_GR_SET − MH_DEL_SET);

(release capabilities on data items not required anymore)

MSS_STEPS = (MSS_STEPS − MH_DEL_STEPS);

MSS_AFF_STEPS = (MSS_AFF_STEPS − MH_DEL_STEPS);

(b) Check if the data read before the last disconnection period is still valid.

i. If (MSS_REV_SET = ϕ ∧ MSS_REQ_SET = ϕ)

- all the requested capabilities on data items accessed before the last disconnection period have been granted and none of these granted capabilities have been revoked. Hence, there is no need for validation of data read earlier or re-execution of any steps at the time of reconnection.
- Go to step (c) to validate the new reads of the last disconnection period.

ii. If (MSS_REV_SET = ϕ ∧ MSS_REQ_SET ≠ ϕ)

- data items on which capabilities were granted have not been revoked during the last disconnection period.
- some of the transaction's requests for specific capabilities have

not yet been granted since the last reconnection point. Though the capabilities with priorities have not yet been issued, the CGM would have issued capabilities with no_priorities to the transaction on these data items. Since the no_priority capabilities were also not revoked in the last disconnection period, the transactions are still reading the latest value of the data items and hence can proceed with the execution in an optimistic fashion.

- Go to step (c) to validate the new reads of the last disconnection period.

iii. If (MSS_REV_SET $\neq \phi \wedge$ MSS_REQ_SET $= \phi$)

- all transaction's requested capabilities were granted, but some of the capabilities were later revoked either due to timeout periods or commitment of other transactions. For each step in the MSS_AFF_STEPS, validate all the read requests which had been revoked and present in MSS_REV_SET.

  Let CT_1^x be the vector timestamp of the data item on which capability was held and CT_2^x the timestamp for the same data item currently available on the MSS.

  CASE 1: A P_RW read capability was granted, but later revoked due to timeout period. In this situation, the CGM

  – validates the data item on which capability was revoked.

  call validate(CT_1^x, CT_2^x).

If **valid** validate the other revoked capabilities of the step.

- If **step is successfully validated**, re-execution is not necessary. Re-request the lost capabilities. The CGM grants no_priority capability if the capability requested cannot be granted right away. Move MSS_AFF_STEPS back to MSS_STEPS. Go to step (c).
- If **not valid**, store the identifiers of this step and any other steps that get affected due to this step as discussed in subsection 5.3.3. Continue validating other steps present in the MSS_AFF_STEPS. After identifying all the steps affected, re-execute the steps in the predefined partial order. **If** results fall within 'acceptance criteria', re-request all the revoked capabilities, go to step (c) **else** go to step (e). On successful re-execution, the steps are moved from MSS_AFF_STEPS to MSS_STEPS.

CASE 2: A P_RW read capability was requested on a data item x. Due to compatibility constraints, the CGM grants NP_RW read capability to the transaction. The NP_RW capability may get revoked by some other transaction which has written a new value in x. Thus, though the requested P_RW capability has not been granted, the transaction that is running in an optimistic fashion needs to be validated or re-executed as discussed in the previous case. The P_RW read capability cannot be granted in this case

as the version on which the priority capability was requested does not exist anymore. The step has to re-executed and only then the capability re-requested on the new version of the data item. Thus, the CGM does not allow priority capabilities on newer versions of data if the transaction had requested the capability on an previous version.

- **Revoking of Capabilities**: Capabilities that are granted to transaction T_i on data items may get revoked either due to expiration of timeout periods or due to commitment of some other transaction T_j. If the capability has been revoked due to commitment of another transaction T_j, some of the steps of the transaction T_i may have to be re-executed. On the other hand if T_i lost the capability with priority on x due to timeout period, then
  - If there is a conflicting capability request by T_j, allow T_j to access the data item x and move the capability request of T_i from its MSS_GR_SET to MSS_REV_SET.
  - If there are no conflicting capability requests waiting, allow T_i to re-obtain the capability and reset it to a second timeout period.
  - If T_j 'timeouts', then T_i can be given the read priority again, but if T_j commits and the capability held was priority on read, then T_i need not re-execute, but if the capability held was priority on write then one or more steps of T_i need to be re-executed at the time of reconnection as validation would have failed forcing re-execution.

iv. If (MSS_REV_SET $\neq \phi \wedge$ MSS_REQ_SET $\neq \phi$)

- some of the transaction's capability requests have not yet been granted and some granted capabilities were revoked in the last disconnection period.
- perform the above two steps (ii) and (iii).
- if successfully validated or re-executed, go to step (c)

  else, go to step (e).

(c) Validate the data accessed in the last disconnection period. Update the following sets:

MSS_REQ_SET = (MSS_REQ_SET $\cup$ MH_NEW_REQ_SET)

i. Validate the new reads in MH_NEW_REQ_SET.
if invalid, re-execute the new steps(MH_NEW_STEPS).
if acceptance criteria satisfied
MSS_STEPS = (MSS_STEPS $\cup$ step re-executed)
re-request the capabilities required by the step;
if transaction execution is complete
go to step (d);
else
continue with transaction execution;
endif
else
Go to step (e);
endif
else
MSS_STEPS = (MSS_STEPS $\cup$ step re-executed)
if transaction is complete;
go to step (d);
else
continue with transaction execution;
endif
endif.

ii. Obtain the requested capabilities in MH_NEW_REQ_SET. The CGM grants no_priority capabilities if priority capabilities cannot be granted.

(d) Transaction execution is complete and has come for final validation. The final validation may be conceptually thought of as conversion of capabilities to locks. The details of this procedure and the correctness proof are discussed in the next section, where in we prove the correctness of the capability granting mechanism. The final validation may involve some waiting. Once a transaction T_i has been successfully validated and committed, and if x has been modified by T_i, then all transactions reading x with no_priority capabilities, that is, transactions present in the CAP_SET will get their read capabilities on x revoked [26].

(e) All the capabilities held by the transaction are revoked and the transaction is re-submitted with new input parameters and acceptance criteria.

When the transaction is either aborted or committed, the MSS empties the various sets and releases all the capabilities held.

5.3.3 Re-execution of steps

Since our execution model employs the concept of an interactive transaction, the revoking of the capabilities held need not require the re-execution of the entire transaction as such but only a few steps. These affected steps are re-executed at the time of reconnection. Let us illustrate this using an airline booking example as shown in the Figure 5.4. In step s_0, the user books a flight from city X to city Y tentatively by flight B. He also associates an acceptance criteria to purchase a ticket on an earlier

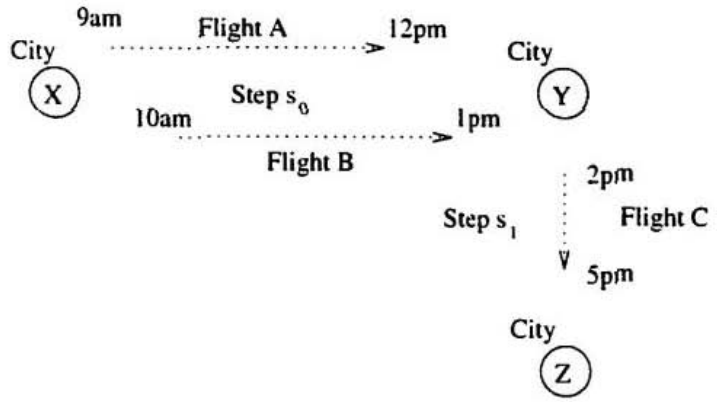


Figure 5.4: Re-executing a single step

flight A, if no seats are available on flight B. He then goes ahead with step s_1 to book the next flight from city Y to Z. On reconnection, the user may have to re-execute step s_0 as read capabilities held by the step may have been revoked. Seats on flight B may no longer be available forcing him to purchase a ticket on flight A. In this case, only step s_0 need to be re-executed as the second flight can still be taken by the user.

Now, let us look at a slightly different scenario in booking the flights as shown in Figure 5.5. User in step s_0 books a seat tentatively on flight A with acceptance criteria to purchase the seat on a later flight B, if no seats are available on the previous flight. In step s_1, the user book a seat tentatively on flight C, with acceptance criteria with acceptance criteria to purchase the seat on a later flight D, if no seats are available on flight C. Now, let us assume that all the seats on flight A are booked, so step s_0 is re-executed and the a seat on flight B booked tentatively. This new booking in turn forces step s_1 to be re-executed as the user will not have time to catch the next flight from city Y. Thus s_0 forces s_1 to be re-executed.

An interactive transaction can be visualized as a partially ordered set of steps as

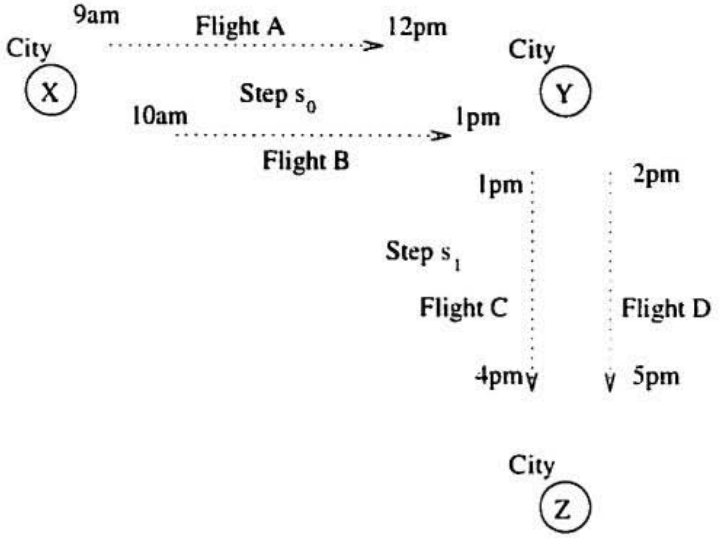


Figure 5.5: Re-executing several steps

shown earlier in the Figure 5.3. From the figure it can be seen that the output of step s_0 triggers two different and independent steps s_1 and s_4. Also, s_3 and s_{10} are independent steps, but s_{11} in-turn depends on the output of s_{10}, and s_9 requires input from both steps s_6 and s_8.

A - $s_0 s_1 s_7 s_8 s_9$

B - $s_0 s_1 s_2 s_5 s_6 s_9$

C - s_3

D - $s_{10} s_{11}$

While A, B, C, and D are totally ordered executions, the transaction as a whole is partially ordered. If a read capability held by step s_0 is revoked and validation fails, then in the *worst case* both ordered executions A and B have to be re-executed. Similarly, if a capability held by s_4 is revoked then $A^1 = s_4 s_7 s_8 s_9$ has to be re-executed. Similarly if a read capability held by

s_7 is revoked and validation fails, $A^2 = s_7 s_8 s_9$ are re-executed,

s_8 is revoked and validation fails, $A^3 = s_8 s_9$ are re-executed,

s_9 is revoked and validation fails, $A^4 = s_9 = B^5$ is re-executed,

s_1 is revoked and validation fails, $B^1 = s_1 s_2 s_5 s_6 s_9$ are re-executed,

s_2 is revoked and validation fails, $B^2 = s_2 s_5 s_6 s_9$ are re-executed,

s_5 is revoked and validation fails, $B^3 = s_5 s_6 s_9$ are re-executed,

s_6 is revoked and validation fails, $B^4 = s_6 s_9$ are re-executed,

s_3 is revoked and validation fails, $C^1 = s_3$ is re-executed,

s_{10} is revoked and validation fails, $D^1 = s_{10} s_{11}$ are re-executed,

s_{11} is revoked and validation fails, $D^2 = s_{11}$ is re-executed.

Thus based on the capability revoked, limited number of steps need to be re-executed instead of the entire transaction. And since we allow for capabilities with priorities, the number of steps that need to be re-executed will also be reduced.

5.4 Correctness Proof

Final Validation: Each transaction on completion of its execution must be validated. Here the validation is different from the intermediate validations during the transaction execution. The final validation can be thought of as conversion of capabilities into 'read' (shared) and 'write locks' (exclusive locks). The point where all the capabilities are validated corresponds to the 'lock point'. The lock point order or the final validation order is the serial order of execution of transactions [19]. Our proof is based on the serial validation scheme in [34].

The data items can be one of the following states based on the capabilities granted:

r^p	r^n	w^{pp}	w^{pn}	w^{np}	w^{nn}	STATE
many	many			1	many	A
	many	1	many		many	B
	many		many	1	many	C

Table 5.8: Allowable capabilities on a data item

The final validation procedure for T_i is as follows:

1. validation of read capabilities for data item x involves the following:

   (a) if capability held is in r^p mode (states A), validation is automatic.

   (b) if capability is in r^n mode, wait until no other transaction holds a write in w^{pp} or w^{pn} modes.

2. validation of write capability for data item x involves the following:

   (a) if capability held is in w^{pp} mode (states B), validation is automatic.

   (b) if capability held is in w^{pn} mode (states B, C), wait until no other transaction holds a write capability on x in P_WW mode,that is, w^{pp} or w^{np} modes.

   (c) if capability held is in w^{np} mode (states A, C), wait until no other transaction has read capability on x in P_RW mode.

   (d) if capability held is in w^{nn} mode, wait until no other transaction is holding r^p, w^{pp}, or w^{np}.

If all 'locks' are obtained by T_i:

- write phase is done. That is, the values written by the transaction are transferred from its private workspace to its database. Invalidate the r^n capabilities

on the data items held by other transactions that were modified by T_i.

- commit the transaction.
- release all the capabilities and locks held.

The point at which all the locks are obtained is the 'lock point', and the lock point order is the effective serialization order of transactions. This follows from the basic two phase locking protocol which ensures serializability [8].

5.5 Handling Deadlock Situations

Deadlocks occur when two transactions are waiting for priority capabilities or locks held by the other to be released. Deadlocks can be handled in many ways. We use timeout mechanism to handle deadlocks. That is, if the requested priority capabilities or locks are not granted to the transaction within this timeout period, the CGM assumes that this transaction is involved in a deadlock and aborts the transaction to break the deadlock.

In our algorithm, deadlock situations can occur at two instances:

1. In the request and acquisition of capabilities phase a situation as shown in Figure 5.6 may lead to a deadlock.

   Since each priority capability is already associated with a timeout to deal with the disconnected MHs, deadlocks will automatically disappear when the timeout period for one of the capabilities expires and hence the capability revoked.

2. The second deadlock situation occurs in the final validation phase. When a transaction comes for final validation, we associate a predefined timeout period.

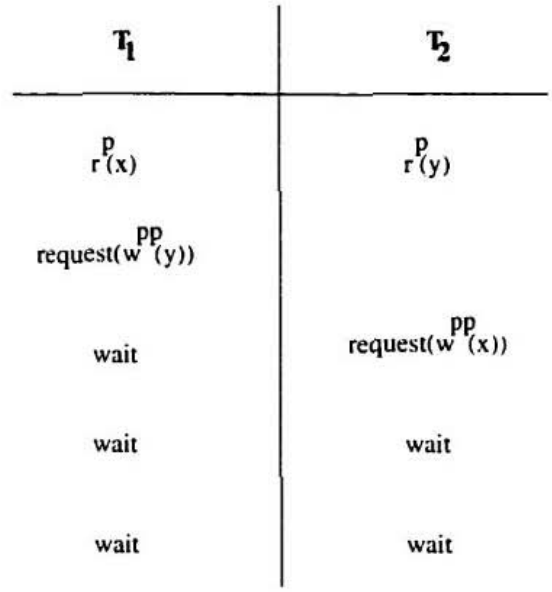


Figure 5.6: Deadlock Situation

The capabilities and locks held by the transaction are not released or revoked until this timeout period expires. The transaction's timeout period nullifies the initial timeout periods assigned to different capabilities individually. In effect, all capabilities are held for the same amount of time once the transaction enters the final validation phase.

Since the CGM is only guessing that the transaction may be involved in a deadlock and revoking all the capabilities, it might be actually making a mistake. Though a transaction does not contribute to a deadlock, it may get aborted for just waiting for a capability held by a transaction which is taking longer to complete. As far as correctness is concerned, there is no harm in making such an incorrect guess. This can be avoided by using long timeout periods. The price to pay would then be that a transaction involved in a deadlock would have

to wait longer before it is actually detected by timing out. Thus the timeout periods are tuned to be long enough so that most transactions that are aborted are actually deadlocked, but short enough that deadlocked transactions don't wait too long for their deadlocks to be noticed.

The timeout periods can be calculated in several ways:

- always fixed.
- based on the number of data items accessed by the transaction.
- based on the number of capabilities held by the transaction.
- based on number of capabilities the transaction is waiting for. etc.

Another approach to detect deadlocks dynamically is to use waits-for graph [8]. The CGM maintains a directed graph which are labeled with transaction names. There is an edge $T_i \rightarrow T_j$, from node T_i to T_j, iff transaction T_i is waiting for transaction T_j to release some capability. If the waits-for graph (WFG) has a cycle, it implies that a transaction is waiting for itself and hence a deadlock. The CGM can check for cycles every time a new edge is added or wait until a few edges are added.

5.6 Mobility

As discussed in the previous chapter, when a MH moves into a new cell we could re-execute the transaction steps on the data available on the new MSS and continue with the execution. If the past execution has been guaranteed in the source MSS by holding priority capabilities, then this execution is still guaranteed as the priorities are obtained based on a global decision. Hence no matter where the MH moves, as far as the guarantees are concerned they are still held till the timeout expires.

5.7 Discussion

In this chapter we presented a flexible integrated concurrency control scheme integrating optimistic and pessimistic approaches to access the data items based on RW- and WW-conflicts for mobile environments. The underlying theme of the transaction execution model is: (i) tentative execution of transactions in the MH: (ii) periodic confirmation of the execution in the MSS and (iii) guaranteeing past execution by switching from optimistic to pessimistic modes (perhaps when the transaction is nearing completion).

In the transaction model proposed, it may appear as if the MH is taxed heavily due to the complexity involved in guaranteeing its past execution. This in reality is not true. The MH can act independently, kept unaware of the capability granting mechanism. All it does is, cache data and do the relevant computation. At some point of time during its execution, it requests for some guarantee for its past execution. It is the Capability Granting Manager on the MSS which acts as an agent to the MH in coordinating the various tasks like granting of capabilities, maintaining timeout periods, revoking capabilities, validation and re-execution of steps, etc.. Also, irrespective of the guarantee being met or not, the transaction on the MH is allowed to proceed. Since most of the workload is shifted to the MSS, the power consumption in the MH is significantly reduced.

Chapter 6

CONCLUSION

This thesis

1. begins with a overview of the design challenges in mobile computing environments.

2. reviews some of the transaction models proposed in the literature for mobile environments.

3. presents a new transaction execution model that facilitates adjusting the computation at the mobile host to the database state at the fixed network. and

4. proposes a method of making the computation at the mobile host more credible by providing partial guarantee against invalidation.

In the transaction execution model. at various intermediate stages the computation at MH is validated. and if validation fails then re-executed at the MSS. One of the implementation issues is to determine the actual stages of validation/re-execution. In this thesis. the stages are taken to be the reconnection points of the MH with the MSS. This of course may not be the best approach. For instance. let us consider a

transaction which is almost complete. That is, say it has already accessed 95% of the data it would require. When the MH reconnects to the MSS to access the other 5%, it is advisable to complete the execution and then validate/re-execute instead of validating all the 95% of the data accessed. So, several criteria like number of data items already accessed, number of data items going to be accessed in the future, type of connections available, applications, tariffs, etc., might influence the choice of determining the intermediate stages.

The enhanced model guarantees the computation at the MH by moving it to pessimistic mode by "locking" some of the data items at some stage during its execution. The effects of locking are significant in mobile environments since mobile hosts (holding locks) may be disconnected for long, unpredictable durations. This is addressed by providing varying degrees of pessimism and timeout periods. This facility is provided by the Capability Granting Manager (CGM) on the MSS and hence places no burden on the MH.

Typically pessimistic approach is advantageous where demand for data is high. On the other hand, optimistic approaches are more suitable in places where the demand for data is low. The disadvantage of this approach is the possibility of invalidation at the end, leading to rollback. In practical situations, demand for data varies with time. Our flexible integrated approach can be tailored to choose pessimistic or optimistic access with respect to the demand.

Bibliography

[1] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast disks: Data management for asymmetric communication environments. In *Proceedings ACM SIGMOD*. San Jose, CA, USA, May 1995.

[2] S. Acharya, M. Franklin, and S. Zdonik. Balancing push and pull for data broadcast. In *Proceedings ACM SIGMOD International Conference on Management of Data*. Tucson, AZ, USA, May 1997.

[3] G. Alonso, R. Günthör, M. Kamath, D. Agrawal, A. El Abbadi, and C. Mohan. Exotica/fmdci: A workflow management system for mobile and disconnected clients. *Distributed and Parallel Databases*, 4(3):229–247, 1996.

[4] R. Alonso and H. F. Korth. Database system issues in nomadic computing. *ACM SIGMOD Record*, 22(2):388–392, 1993.

[5] D. Barbara and T. Imielinski. Sleepers and workaholics: Caching strategies in mobile environments. *ACM SIGMOD Record*, 23(2):1–12, 1994.

[6] D. Barbara, R. Jain, and N. Krishnakumar. *Databases and Mobile Computing*. Kluwer Academic Publishers, Boston, 1996.

[7] P. Bernstein and N. Goodman. Timestamp based algorithms for concurrency control in distributed database systems. In *Proceedings 6th Very Large Data Base Conference*, 1980.

[8] P. Bernstein, Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Co., 1987.

[9] Bharat K. Bhargava. *Concurrency Control and Reliability in Distributed Systems*. Van Nostrand Reinhold Company Inc., New York, 1987.

[10] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[11] Y. Breitbart, H. Garcia-Molina, and Silberschatz. Overview of multidatabase transaction management. *VLDB*, 1:1–39, 1992.

[12] O. Bukhres, S. Morton, E. Vanderdijs, P. Zhang, C. Crawley, J. Platt, and M. Mossman. A proposed mobile architecture for distributed database environment. In *Proceedings 5th Euromicro Workshop on Parallel and Distributed Computing*, 1997.

[13] Omran A. Bukhres and Ahmed A. Elmagarmid. *Object-Oriented Multidatabase Systems: A Solution for Advanced Applications*. Prentice Hall Inc., New Jersey, 1996.

[14] P. K. Chrysanthis. Transaction processing in a mobile computing environment. In *Proceedings IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 77–82, 1993.

[15] P. K. Chrysanthis and K. Ramamritham. Acta: A framework for specifying and reasoning about transaction structure and behavior. In *Proceedings ACM SIGMOD*, pages 194–203, 1990.

[16] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch. The bayou architecture: Support for data sharing among mobile users. In *Proceedings IEEE Workshop on Mobile Computing Systems and Applications, Santa Cruz, CA*, pages 2–7, 1994.

[17] M. H. Dunham, A. Helal, and S. Balakrishnan. A mobile transaction model that captures both the data and movement behavior. *ACM/Baltzer Journal on Special Topics in Mobile Networks and Applications (MONET)*, 2:149–162, 1997.

[18] Ahmed K. Elmagarmid. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, Inc., 1995.

[19] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11), 1976.

[20] G. H. Forman and J. Zahorjan. The challenges of mobile computing. Technical Report 93-11-03, Computer Science and Engineering Department, University of Washington, Washington, USA, March 1994.

[21] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings ACM SIGMOD*, pages 249-259, 1987.

[22] V. Gligor and R. Popescu-Zeletin. Transaction management in distributed heterogeneous database management systems. *Information Systems*, 11:287–297, 1986.

[23] J. Gray. *In Operating Systems, An Advanced Course*. Springer-Verlag, New York, 1979.

[24] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings ACM SIGMOD*, pages 173–182, 1996.

[25] R. Gruber, F. Kaashoek, B. Liskov, and L. Shrira. Disconnected operation in the thor object-oriented database system. In *Proceedings IEEE Workshop on Mobile Computing Systems and Applications, Santa Cruz, CA*, pages 51–56, 1994.

[26] U. Halici and A. Dogac. An optimistic locking technique for concurrency control in distributed databases. *IEEE Transactions on Software Engineering*, 17(7):712–724, 1991.

[27] G. Herman, G. Gopal, K. Lee, and A. Weinrib. The data cycle architecture for very high throughput database systems. In *Proceedings ACM SIGMOD*, May 1987.

[28] A. D. Joseph, J. A. Tauber, and M. F. Kaashoek. Mobile computing with the rover toolkit. *IEEE Transactions on Computers*, 46(3):337–352, 1997.

[29] G. E. Kaiser. A flexible transaction model for software engineering. In *Proceedings 6th International Conference on Data Engineering, Los Angeles, CA*, pages 560–567, 1990.

[30] J. J. Kistler and M. Satyanarayanan. Disconnected operations in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

[31] J. Klingemann, T. Tesch, and J. Wäsch. Enabling cooperation among disconnected mobile users. In *Proceedings Second IFCIS International Conference on Cooperative Information Systems*, 1997.

[32] H. Koch, L. Krombholz, and O. Theel. A brief introduction into the world of 'mobile computing'. Technical Report THD-BS-1993-03, Computer Science Department, University of Darmstadt, Darmstadt, Germany, May 1993.

[33] S. Krishna, N. H. Vaidya, and D. K. Pradhan. Recovery in distributed mobile environments. In *Proceedings IEEE Workshop on Advances in Parallel and Distributed Systems*, 1993.

[34] H. T. Kung and J. T. Robinson. Optimistic methods for concurrency control. *ACM Transactions on Distributed Systems*, 6:213–226, 1981.

[35] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, 1992.

[36] Q. Lu and M. Satyanarayanan. Isolation-only transactions for mobile computing. *Operating Systems Review*, pages 81–87, April 1994.

[37] Q. Lu and M. M. Satyanarayanan. Improving data consistency in mobile computing using isolation only transactions. In *Proceedings 5th Workshop on Hot Topics in Operating Systems*, 1995.

[38] S. K. Madria and B. Bhargava. A transaction model to improve data availability in mobile computing. Technical report, Nanyang Technical University, 1997.

[39] K. A. Momin and K. Vidyasankar. A model for transaction execution in mobile environments. In *Proceedings International Conference on Information Technology. India*, pages 162–167, Bhubaneswar, India, 1998.

[40] W. Montgomery. Robust concurrency control for distributed information system. Technical Report MIT/LCS/TR-207, MIT Laboratory for Computer Science, Cambridge, MA, USA, December 1978.

[41] S. Morton and O. Bukhres. Mobile transaction recovery in distributed medical databases. In *Proceedings 8th International Conference on Parallel and Distributed Computing and Systems*, 1997.

[42] J. E. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, 1981.

[43] V. R. Narasayya. Distributed transactions in a mobile computing system. Technical report, Computer Science and Engineering Department, University of Washington, Washington, USA, March 1994.

[44] M. Nodine and S. Zdonik. Cooperative transaction hierarchies: A transaction model to support design applications. In *Proceedings International Conference on Very Large Data Bases*, pages 83–94, 1984.

[45] Patrick E. O'Neil. The escrow transactional model. *ACM Transactions on Database Systems*, 11:405–430, December 1986.

[46] E. Pitoura and B. Bhargava. Maintaining consistency of data in mobile distributed environments. In *Proceedings 15th International Conference on Distributed Computing Systems*, pages 404–413, 1994.

[47] E. Pitoura and B. Bhargava. Revising transaction concepts for mobile computing. In *Proceedings IEEE Workshop on Mobile Computing Systems and Applications, Santa Cruz, CA*, pages 164–167, 1994.

[48] E. Pitoura and G. Samaras. *Data Management for Mobile Computing*. Kluwer Academic Publishers, Boston, 1998.

[49] C. Pu, G. E. Kaiser, and Hutchison. Split-transactions for open-ended activities. In *Proceedings 14th Very Large Data Base Conference*, 1988.

[50] A. Rasheed and A. Zaslavsky. Ensuring database availability in dynamically changing mobile computing environment. In *Proceedings 7th Australian Database Conference*, pages 100–108, 1996.

[51] A. Rasheed and A. Zaslavsky. A transaction model to support disconnected operation in a mobile computing environement. In *Proceedings 4th International Conference on Object Oriented Information Systems*, 1997.

[52] A. Rester. Contract: A means for extending control beyond transaction boundaries. In *Proceedings 2nd International Workshop on High Performance Transaction Systems*, 1989.

[53] D. Rosenkrantz, R. Stearns, and Lewis P. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems*, 3(2), June 1978.

[54] M. Rusinkiewich, W. Klas, T. Tesch, J. Wäsch, and P. Muth. Towards a cooperative transaction model: The cooperative activity model. In *Proceedings 21st International Conference on Very Large Database Systems*, pages 194–205, 1995.

[55] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Proceedings 15th ACM Symposium on Principles of Distributed Computing*, pages 1–7, May 1996.

[56] M. Satyanarayanan. Mobile information access. *IEEE Personal Communications*, 3(1), 1996.

[57] G. Schlageter. Optimistic methods of concurrency control in distributed database systems. In *Proceedings 7th International Conference on Very Large Database Systems*, 1981.

[58] M. D. Schroeder, A. D. Birrell, and R. M. Needham. Experience with grapevine. *ACM Transactions on Computer Systems*, 2:3–23, 1984.

[59] M. Sinha and et al. Timestamp based certification schemes for transactions in distributed database systems. In *Proceedings ACM SIGMOD*, Austin, Texas, USA, May 1985.

[60] J. D. Solomon. *Mobile IP - The Internet Unplugged*. Prentice Hall Inc., New Jersey, 1998.

[61] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall Inc., 1991.

[62] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, 1996.

[63] Jian Tang. On multilevel voting. *Distributed Computing*, 8:39–58, 1994.

[64] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems*, pages 140–149, 1994.

[65] K. Vidyasankar. Generalized theory of serializability. *ACTA Informatica*, 24:105–119, 1987.

[66] K. Vidyasankar and Vijay V. Raghavan. Highly flexible integration of the locking and the optimisitic approaches of concurrency control. In *Proceedings IEEE 9th COMPSAC Computer Software and Applications Conference*, pages 489–494, 1985.

[67] G. Walborn and P. K. Chrysanthis. Pro-motion: Support for mobile data access. *Personal Technologies*, 1(3):171–181, 1997.

[68] G. D. Walborn and P. K. Chrysanthis. Supporting semantics-based transaction processing in mobile database applications. In *Proceedings 14th IEEE Symposium on Reliable Distributed Systems*, 1995.

[69] J. Wäsch and W. Klas. History merging as a mechanism for concurrency control in cooperative environments. In *Proceedings RIDE-Interoperatbility of Nontraditional Database Systems*, pages 76–85, 1996.

[70] L. H. Yeo and A. Zaslavsky. Submission of transactions from mobile workstations in a cooperative mutidatabase processing environment. In *Proceedings 14th International Conference on Distributed Computing Systems*, pages 372–379, 1994.

[71] S. Zdonik, M. Franklin, R. Alonso, and S. Acharya. Are 'disks in the air' just pie in the sky? In *Proceedings IEEE Workshop on Mobile Computing Systems and Applications, Santa Cruz, CA*, pages 12–19, 1994.