A DYNAMIC DATA/CURRENCY PROTOCOL FOR MOBILE DATABASE DESIGN
AND RECONFIGURATION

By

YANLI XIA

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2002
Copyright 2002

by

Yanli Xia
ACKNOWLEDGMENTS

I express my sincere gratitude to my advisor, Dr. Abdelsalam (Sumi) Helal, for giving me the opportunity to work on this challenging topic and for providing continuous guidance during my thesis writing. I am thankful to Dr. Joachim Hammer and Dr. Herman Lam for agreeing to be on my supervisory committee.

I really appreciate Youssef Kaddoura’s help in proofreading my thesis. I also thank the other members in the Harris lab for their help.

I would like to take this opportunity to thank my parents, my husband and my son, for their continued and encouraging support throughout my period of study and especially in this endeavor.
# TABLE OF CONTENTS

ACKNOWLEDGMENTS..................................................................................................................iii

ABSTRACT........................................................................................................................................vi

CHAPTER

1 INTRODUCTION.............................................................................................................................1

1.1 Challenges in Mobile Databases .........................................................................................1
1.2 Related Work.........................................................................................................................3
1.3 Organization of the Thesis .................................................................................................5

2 A MODEL FOR MOBILE DATABASES......................................................................................6

2.1 Interface Assumption ...........................................................................................................8
2.2 Scenarios of Mobile Hosts and Fixed Hosts .......................................................................8
   2.2.1 Connection Scenarios Between A Mobile Host and A Fixed Host.........................8
   2.2.2 Connection Scenarios Between Mobile Hosts .......................................................9
2.3 Scenarios of Ad-hoc Network to Fixed Network .............................................................10
2.4 A Model for Mobile Databases .........................................................................................11

3 GENERALIZED HOARDING AND SYNCHRONIZATION MECHANISMS .......................15

3.1 Background .......................................................................................................................16
3.2 Data and Currency Hoarding Modes .................................................................................17
   3.2.1 Currency Hoarding Modes ....................................................................................17
   3.2.2 Data Hoarding Modes ..........................................................................................20
3.3 Currency and Data Synchronization Modes .......................................................................22
   3.3.1 Currency Synchronization Modes .........................................................................23
   3.3.2 Data Synchronization Modes ...............................................................................25
3.4 Hoarding and Synchronization Algorithms ......................................................................27
   3.4.1 Definition of Work Sets .......................................................................................27
   3.4.2 Hoarding Algorithms ............................................................................................29
   3.4.3 Synchronization Algorithms ...............................................................................30
4 METADATA MANAGEMENT FOR MOBILE DATABASES .................................. 34

4.1 Categories of Metadata in Mobile Databases ........................................... 35
   4.1.1 Connectivity ......................................................................................... 35
   4.1.2 Availability .......................................................................................... 36
   4.1.3 Access .................................................................................................. 37
   4.1.4 User-defined Constraints ..................................................................... 37

4.2 Metadata Collection in Mobile Databases ............................................... 37
   4.2.1 Focused Subset of Metadata ................................................................ 37
   4.2.2 Metadata Collection ............................................................................ 39

5 DYNAMIC CURRENCY ALLOCATION AND REDISTRIBUTION .............. 44

5.1 A Target of Currency Allocation ............................................................... 44
5.2 Dynamic Currency Redistribution Referring to Metadata ....................... 46
   5.2.1 Correctness Requirements of Currency Redistribution ..................... 46
   5.2.2 Currency Redistribution .................................................................... 47

5.3 Ad-hoc Database Check-out/Check-in Currency ...................................... 51
5.4 Creation and Retirement of Currency and Data ........................................ 52

6 SIMULATION .................................................................................................... 54

6.1 Simulation Assumption ............................................................................. 54
6.2 Simulation Model ........................................................................................ 55
6.3 Simulation Settings ..................................................................................... 57
6.4 Simulation Results ...................................................................................... 59
   6.4.1 Commit Rate and Commit Percentage ............................................... 59
   6.4.2 Mean Commit Delay .......................................................................... 62

CONCLUSIONS AND FUTURE WORK ............................................................. 64

LIST OF REFERENCES ....................................................................................... 66

BIOGRAPHICAL SKETCH .................................................................................. 69
Mobility is changing the way we design databases and their Database Management Systems (DBMS). For example, the scenario in which all database users are ad-hoc users has been integrated in the Bayou and Deno projects. Both of them take an asynchronous, epidemic information flow and conflict avoidance-based (i.e., pessimistic) approach to ensure that all committed updates are serialized in the same order at all servers. Bayou uses a primary-copy scheme, while Deno takes a bounded weighted-voting one.

In this thesis, we focus on flexible protocols for dynamic database design and reconfiguration. In particular, we introduce protocols and ideas that enable databases to be designed in such a way that their location, replication and even use semantics can be changed dynamically. We believe the flexibility of this approach is necessary given the uncertainty of user mobility.
We introduce a dual data/metadata hoarding and synchronization protocol. We are especially focused on the currency type of metadata, which is a weight (vote) given to a replica. Data are hoarded via a variety of hoarding mechanisms with different hoarding semantics. Metadata record the changing characteristics of mobile hosts and currency is dynamically redistributed to fit the target. Primaries are allowed to be diluted, concentrated and transferred in the system. We borrow a bounded weighted-voting scheme from Deno and use it to do currency synchronization. We also use anti-entropy mechanisms to do data synchronization to eventually bring the database to a consistent state.

We include a fixed network as the home of the database. An ad-hoc network checks out its desired data and checks them back in when they are no longer accessed by the mobile users.

Finally, we conduct experimental evaluation using simulation to assess the benefits of our protocols.
CHAPTER 1
INTRODUCTION

1.1 Challenges in Mobile Databases

Since their emergence in the 1970s, wireless networks have become increasingly popular in the computing industry. This is especially true within the past decade, which has seen wireless networks being adapted to enable mobility. Mobile computing is the merger of advances in portable computing and wireless communication with the aim of providing seamless and ubiquitous services for mobile users. In mobile environments, database applications are enhanced with the useful features of wireless technology. However, mobile computing environments have severe resource constraints and unstable operating conditions. Compared to the wired network, the wireless medium is notable for its high cost, narrow bandwidth and low reliability. The wireless connection is usually weak and intermittent, causing frequent disconnection. Mobile hosts are usually resource poor, battery dependent, and susceptible to accidental damage or loss.

Mobility is changing the way we design databases and their DBMS. Many software problems associated with data management, transaction management, and data recovery have their origin in distributed database systems. In mobile computing, however, these problems become more difficult to solve. Large-scale replication is provided to allow anytime/anywhere access. Data interests may change with shifting user context, so a new data management decision is needed.

Due to the characteristics of mobile environment, mobile computing exposes key limitations of the traditional computing model [4]:
1) Weak connectivity/frequent disconnection: state of wireless communication and cost issues.

2) Large-scale replication: device-local hoarding required due to availability when disconnected.

3) Close user interaction/feedback: allows negotiation, partial and preliminary results.

4) Long-running tasks: always a problem for ACID systems

5) Real-time constraints

There are some projects related to databases in ad-hoc networks, such as Bayou [9, 23] and Deno [5, 6, 7, 16]. But homes of databases are usually fixed network. For example, members of a traveling salesman group may dial their head office to get new product information and put sales and order data into the company’s home database; a botanist may wish to put her new sighting data into the laboratory database so that others may see it before her return; a UPS delivery needs to report the package information to the home office to make it available to the online searching system for customers.

In any situation in which multiple databases in ad-hoc networks are used or in which there is a home-base database, we require replication to synchronize the data and operations between these databases. Users’ interests in data may change shifting context from time to time, place to place, so we need to track user’s state, based on position, time, history and workflow to form metadata to adapt to the new user context.

In this paper, we focus on flexible protocols for dynamic database design and reconfiguration. We introduce a dual data/currency hoarding and synchronization protocol. Data are hoarded via P-, G-, or C-hoarding and accompanying hoarded currency indicates semantics of data. Metadata record the changing characteristics of mobile hosts and currency is dynamically redistributed to fit the target. Databases are designed in such
a way that their location, replication and even use semantics can be changed dynamically. We believe the flexibility of this approach is necessary given the uncertainty of user mobility. We include fixed network as home of database. Finally, we conduct experimental evaluation using simulation to assess the benefits of our protocols.

1.2 Related Work

Many mobile computing issues associated with data management, transaction management, and data recovery have their origin in distributed database systems. A distributed database is defined as a collection of multiple, logically interrelated database distributed over a computer network [22]. The existing distributed database systems are extended to support mobile operations (e.g., disconnection, caching, reconciliation, etc.).

The work [12] of Jim Gray et al. showed that replication protocols go along two dimensions: master/group object ownership and lazy/eager update propagation. For master ownership, only primary copy can be updated, while for group ownership, any replica can be updated. Eager update propagation requires all replicas updated in a single transaction, and the lazy one propagates updates asynchronously. They argued that eager schemes are unsuitable for mobility due to frequent disconnections and proposed a two-tier lazy protocol for scalable replication. The solution approaches are as follows: 1) Multiple tiers of hosts: the high connectivity and resource-rich inner ring, land weakly connected, mobile, expendable clients. 2) Two classes of copies: servers retain “copies of record” and clients cache secondary (“soft-state”) copies. Reads see weaker-consistency (snapshot isolation) and updates happen without two-phase commit. Synchronization metadata are kept at clients and servers.

Jajodia and Mutchler [15] and Jajodia et al. [29, 30, 31] improved Jim Gray’s approach using an adaptive data replication algorithm, which changes the replication
scheme of the object as changes occur in the read-write pattern of the object. The algorithm continuously moves the replication scheme towards an optimal one, but this algorithm is designed for a distributed, wholly interconnected information environment, which is not proper for mobile environment.

Lee and Helal [19] introduced a new mobile transaction model (HiCoMo) applicable to decision-making applications over aggregate data warehoused on mobile hosts. The model allows the aggregate data to be updated in disconnection mode, while guaranteeing a very high rate of commitment on reconnection.

Epidemic information flow [2, 8] is used to propagate updates and drive the replicas toward consistency. Epidemic algorithms require few guarantees from the underlying communication system, hence proper to be used in mobile environments. Anti-entropy is an example of epidemic processes, exchanging database content with a peer to resolve any differences between them.

CODA [18, 21] is a distributed file system with the ability to support disconnected operation for mobile computing. It employs two-tier data replication to get scalability: server replication and client side persistent caching of files. Write back caching is the scheme to reach consistency.

SEER [17] is a predictive hoarding system for disconnected mobile operation. Briefly, it observes the user's file-access patterns while a mobile computer is connected to the network, and uses those to predict what files will be needed when the network is no longer available. It then arranges to have those files stored on the mobile machine in advance of disconnection.
Bayou [9, 23] uses epidemic information flow via anti-entropy sessions. It is a more pessimistic approach and ensures that all committed updates are serialized in the same order at all servers using a primary-copy scheme. Updates can only be committed at the primary-copy server.

Deno [5, 6, 7, 16] takes an asynchronous bounded weighted-voting scheme via epidemic information flow such that updates are committed in a decentralized fashion. Deno uses fixed per-object currencies for voting and data movement requires only pairwise communication. Each replica is assigned a currency and total currency in the system is bounded.

1.3 Organization of the Thesis

In chapter 2, we present a model for mobile databases that integrates previously proposed ideas such as ad-hoc and broadcast disks. Then we introduce data and currency hoarding and synchronization in chapter 3. In chapter 4, we give the categories of metadata and work on algorithms to manage metadata. Chapter 5 gives the details of a dynamic currency redistribution protocol based on the metadata. We then present some experiments to evaluate the merits of our protocol in chapter 6. Finally, we conclude our work and list future work in chapter 7.
CHAPTER 2
A MODEL FOR MOBILE DATABASES

Traditional database design is static. The question being addressed is how the database and applications that run against it should be placed across the sites [22]. There are two basic alternatives to placing data: partitioned (non-replicated) and replicated. Respectively, the two fundamental design issues are fragmentation and distribution. No matter which scheme is provided, however, it is static from the angle of the database design. Once data placing is determined, it will not be changed. A directory contains information about data items in the database is maintained. The definition of fragments and their placement determine the contents of the directory. Directory placement and contents influence concurrency control strategies as well as the processing of queries. With directory, data must be located first. The static database design limits flexibility of database applications.

Many mobile computing issues associated with data management, transaction management, and data recovery have their origin in distributed database systems. However, mobility of mobile hosts on the network causes static data in stationary networks to become dynamic and volatile in wireless networks. In mobile database everything is dynamic, varying from sporadic accesses by individual users to particular data to continuous access of a particular data by a large group of user. This is the case from disconnected database access to broadcast disks. For broadcast data, due to the mobility of the mobile hosts, the content of any data to be broadcast should be dynamic and adaptive. Moreover, in an ad-hoc network, the users group and data interests are
changing from time to time. Data partition, location and replication are always dynamic. These characteristics raise interesting issues for the design of mobile databases.

CODA [18, 21] is a distributed file system with the ability to support disconnected operation for mobile computing. The Coda model is that there are many clients and a few servers. Only Coda clients can be used in disconnected and weakly connected mode to support mobile computing. Coda employs server replication and persistent client side caching of files. Directories and attributes are maintained for high performance. Moreover, file-system objects distribution in servers is static in Coda.

Broadcast disk [4, 1, 27] is the proactive distribution of relevant data to large number of users. In order to match data/events to user interests and distribute the data to users, reliable delivery especially with movement and disconnection, broadcast scheduling should be dynamic to satisfy the users and adaptive to the changing context.

The scenario where all database users are ad-hoc users has been integrated in the Bayou and Deno projects. Both of them take asynchronous, epidemic information flow and conflict avoidance-based (i.e., pessimistic) approach to ensure that all committed updates are serialized in the same order at all servers. Bayou uses a primary-copy scheme, while Deno takes a bounded weighted-voting one.

Although there are above significant works on database issues, there is a need to reach a generic solution for dynamic mobile database design to combine distributed databases, broadcast disks and ad-hoc databases. Here we focus on flexible protocols for dynamic database design and reconfiguration involving fixed network. First we give connection scenarios of mobile hosts and fixed hosts, ad-hoc network to fixed network,
then we introduce the concept of mobile databases and specify the configuration used in our system.

2.1 Interface Assumption

A mobile host may have four network interfaces:

1) Strong connect interface (dock);
2) Weak connect interface (wireless);
3) Wireless ad-hoc network interface;
4) Broadcast receiver;

Interfaces 2 and 3 are different in that interface 2 is used to connect with fixed hosts, while interface 3 is used to connect with other mobile hosts in ad-hoc network.

2.2 Scenarios of Mobile Hosts and Fixed Hosts

In this section, the connection scenarios among mobile hosts and fixed hosts are given.

2.2.1 Connection Scenarios Between A Mobile Host and A Fixed Host

Figure 2-1 Connection scenarios of a mobile host with a fixed host
A mobile host may be strong connected (docked) with a fixed host when the mobile host backs “home”. A docked mobile host is equivalent to a fixed host in that it has access to the resources in the fixed network. Besides, it may perform hoarding directly from fixed hosts.

A mobile host may be weak connected (wireless) with a fixed host when there is no existing wired connection, providing that it has wireless network connection interface. Due to cost and battery life, weak connection is normally periodic, lasting not much long.

Most of the time, a mobile host is disconnected from the fixed network, working on local data hoarded before planned disconnection. For unexpected disconnection, a mobile host may seek to reconnect with a fixed host after failure recovery.

When a mobile host is strong connected, we say the mobile host is *homed*, otherwise it is said *mobile*.

### 2.2.2 Connection Scenarios Between Mobile Hosts

A mobile host may be weak connected with or disconnected from another mobile host. We say such mobile hosts are *peers*. Peer mobile hosts may communicate pair-wise such that each connection session only involves two peers.

![Diagram](image)

*Figure 2-2 Connection scenarios between mobile hosts*
When a mobile host is weak connected with one of its peers in the ad-hoc network, they may perform hoarding to get desired data, synchronize their updates, anti-entropy to eliminate their differences, etc. But all these should be done in an efficient manner.

A mobile host is disconnected from its peer after information exchanges. Then it goes on working based on its local data until next connection.

2.3 Scenarios of Ad-hoc Network to Fixed Network

As shown in the figure 2-3, an ad-hoc network may be all docked, half-docked or all undocked to the fixed network.

![Figure 2-3 Connection scenarios of ad-hoc with fixed network](image-url)
When all mobile hosts in an ad-hoc network back home, it is said the ad-hoc network is *docked*. A docked ad-hoc network joins the fixed network to form a bigger distributed system except that it performs hoarding before next mobile.

If some of mobile hosts in an ad-hoc network back home and the others do not, it is said the ad-hoc network is half-docked. The docked mobile hosts are strong connected with the fixed network, while the others are weak connected. Moreover, the weak connected mobile hosts may get information from a broadcast server in the fixed network. The broadcast server continually broadcasts data that mobile hosts are commonly interested in. This way is called pushing. A pushing channel usually has good bandwidth.

The scenario where all mobile hosts in the ad-hoc network weak connected with or disconnected from the fixed network is considered as all undocked. In this case, all mobile hosts leave home and they may perform pair-wise connection to exchange information as well as communicate with fixed hosts. Additionally, ad-hoc network may get pushed data from broadcast.

### 2.4 A Model for Mobile Databases

A mobile host in mobile may communicate with only fixed hosts, no connection available between mobile hosts. In this case, data items reside in all such mobile hosts form a *disconnected database*. This configuration introduces the concept of ubiquitous data access.

A mobile host may wish to communicate with other mobile hosts selected randomly or according to some constraints. Pair-wise connection is enough for such case. This gives the concept of *ad-hoc databases*. A mobile host in an ad-hoc database may connect with fixed hosts to do hoarding or synchronization.
A broadcast server may be integrated into the fixed network and broadcast data that mobile hosts have common interests in. This introduces the concept of broadcast databases. A broadcast database usually contains public information such as weather, stock and financial reports. Mobile hosts grab such information from the air.

Traditionally, database distributed in a fixed decentralized network is a distributed database. We envision that a mobile database should be defined as the union of disconnected database, broadcast disks, ad-hoc database and distributed database. This is because the dynamic nature of the mobile environment may require data to be reallocated or replicated. Mobile environment also may require users and replicas to be ad-hoc or disconnected from the network. Figure 2-4 shows the concept of mobile database.

![Figure 2-4 Mobile database: a whole image](image-url)
For distributed database, there is significant body of previous work [22]. As to

disconnected database, there are several projects on ubiquitous database access working
on it [13, 20]. There are research efforts in broadcast disks [4, 1, 27]. Bayou and Deno do
great work in ad-hoc databases. However, as far as we know, there is no work toward the
whole image of mobile databases. In this thesis, our attention is focused on the scope of
ad-hoc database and its interaction with distributed database. When the mobile hosts in
the ad-hoc network are all docked into the distributed network, they can communicate
with the fixed hosts in the distributed network and be treated as part of it. Our focus is on
the ad-hoc network half-docked or all-undocked cases, in which databases need to be
reconfigured and redesigned to adjust to the dynamic characteristics of the new mobile
environment. When the ad-hoc network finished its work, it may return to the distributed
network and become all docked again.

A motivating example. Ad-hoc users and fixed users may have common interests
in some data. For example, IBM salespeople Frank, Joe, and Nancy collectively cover the
state of Texas, they might expect to be able to consolidate their sales data when they meet
in Austin. While their manager in New York, Ronald, would like to know the results and
decide sales in Florida. Bonnie, another manager in New York, needs the sales data to
make a financial report. In this case, Frank, Joe, and Nancy may wish to hoard their
interested data items and semantics that are able to indicate the primaries. They form an
ad-hoc database and since they have the primary semantics, they may make updates
commitment decision. Ronald and Bonnie are members of distributed database. They
may get information of the up-to-date data from ad-hoc database, or may update the data
items they have and inform ad-hoc users to see if the updates are acceptable. After the
three-people team back, they return the primary to distributed database, i.e. primary is back home.
CHAPTER 3
GENERALIZED HOARDING AND SYNCHRONIZATION MECHANISMS

The mobile computing environment contains large number of low powered mobile hosts. The mobile hosts will often be disconnected due to power limitations, inaccessible communication channels, or as mobile hosts move between different cells. Replication will be essential in such environment, increasing data availability to mobile users: when a mobile host is disconnected, it can continue to process data stored locally. At the same time, replicated data can improve performance. In a mobile environment, it is important to have dynamic replicated data management algorithms that allow for the database to be reconfigured due to the dynamic nature of mobile environments. We introduce a dual data/currency hoarding and synchronization protocol to deal with replication and consistency issues in mobile databases.

An ideal replication scheme would achieve four goals [12]:

1) Availability and scalability: Provide high availability and scalability through replication, while avoiding instability.

2) Mobility: Allow mobile nodes to read and update the database while disconnected from the network.

3) Serializability: Provide single-copy serializable transaction execution.

4) Convergence: Provide convergence to avoid system delusion.

We take a dual data/currency hoarding and synchronization approach. Data hoarding is done by mobile or fixed hosts to hoard data requested by their applications. Currency hoarding accompanies data hoarding and the amount of currency indicates
semantics of data. Currency synchronization is a weighted voting process to collect plurality of currency for an update to commit. Data synchronization is performed by anti-entropy information flow, propagating commitment updates in the system.

3.1 Background

The concept of currency is introduced by research efforts of [26, 28, 32]. Deno [5, 6, 7, 16] uses currency extensively to data replication and voting scheme in mobile environments. What is currency? Simply, currency is a number associated with each replica as a form of priority. The amount of currency held by a given replica is used as that replica’s weight during voting rounds.

Anti-entropy is important to avoid system delusion. The goal of anti-entropy is for two replicas to bring each other up-to-date. In Bayou [9, 23], the storage system at each replica consists of an ordered log of writes and a database that results from the in-order execution of these writes. Anti-entropy needs to enable two replicas to agree on the set of writes stored in their log. In Deno, election information flows from voter to voter through anti-entropy sessions. An anti-entropy session is a uni-directional flow of information specifying elections that have been won, and votes in the current election.

CODA [18, 21] is a distributed file system with the ability to support disconnected operation for mobile computing. SEER [17] is a predictive hoarding system. Bayou and Deno projects concern only pure ad-hoc scenarios. There is no general hoarding and synchronization mechanisms working on mobile databases.

In this chapter, we try to give the generalized hoarding and synchronization mechanisms. The purpose of the mechanisms is to drive data and currency to flow in the mobile databases to dynamically design and reconfigure it.
3.2 Data and Currency Hoarding Modes

Why hoarding? For a mobile host, connection is expensive in that it expends bandwidth, battery life, hand off, etc. Moreover, planned and unexpected disconnections are frequent, while some applications require access to data items anywhere/anytime/anyhow. Lack of access to a data item may halt work on a particular task or even make the computer unusable. Therefore, hoarding is very necessary to get good data availability in that requested data items are saved on the local storage prior to disconnection. We use a dual data / currency hoarding scheme, in which semantics of data is indicated by the amount of currency. Data and currency flow in the system to meet different needs of different users. The configuration of the database is dynamic.

3.2.1 Currency Hoarding Modes

Any shared data item in mobile databases is replicated as a primary and a set of copies, i.e. logical data $X = a$ physical primary of $X + a$ number of physical copies of $X$. The primary and copies all hold some amount of currency and may reside in any hosts. The summary of currencies for each data is fixed: 100. Every data hoarding is accompanied with currency hoarding. The amount of currency held by a given primary or copy is used as the weight of that primary or copy during voting rounds. Additionally, the amount of a data’s currency indicates the semantics of the data, defined as following:

1) Primary currency: If the amount of currency is greater than 50, it indicates that the replica is a primary. A primary has the majority of currency and hence has the power to commit an update.

2) Updateable-copy currency: If the amount of currency is greater than 0 but less than 50, the copy holding such currency is an updateable copy. An updateable copy may propose a local update and start an election to collect plurality of currency votes to commit the update.
3) Read-only copy currency: If the amount of currency is equal to 0, the replica is a read-only copy. It may be involved in the anti-entropy progress to get the up-to-date copy of the data, but it is not allowed to issue an update locally and it has no voting weight either.

4) Undefined currency: If there is no definition of currency for a data item, it means a host does not hold any copy of the data at all. An undefined currency is represented as “⊥”.

A copy currency is either updateable-copy currency or read-only copy currency. There is no negative currency.

An object’s currencies are flowed in the network via pair-wise currency hoardings. The total amount of currencies is fixed at any time. There’re three modes of currency hoarding corresponding to data hoarding: currency G-hoarding, currency P-hoarding, and currency C-hoarding.

• Currency G-hoarding

Before hoarding, the sender holds a primary currency. After $b (0 \leq b < a)$ currency goes to the receiver, the sender still owns a primary currency. Since the total currency for each object is fixed, it’s definitely that the receiver owns a copy currency.

• Currency P-hoarding

Before hoarding, the sender holds a primary currency. After $b (b \geq 50 - c)$ currency goes to the receiver, the sender loses the primary currency and only holds a copy currency. The receiver now gets majority of currency and becomes the new primary holder.
• Currency C-hoarding

Before hoarding, the sender holds a copy currency. After $b \leq a < 50$ currency goes to the receiver, the sender holds a copy currency. If $b$ is equal to $a$, then the copy in the sender becomes read-only. The receiver now gets more currency.

There are three scenarios involving primaries that worth being noticed: primary dilution, primary concentration and primary transfer.

• Primary dilution:

For a sender with a primary currency, if a currency hoarding-out results in a less than 50 currency left, we say the primary is *diluted* at the sender.

• Primary concentration:
For a receiver with a copy currency, if it collects a primary currency after a currency hoarding, as we show above in currency C-hoarding, if \((c + b) > 50\), we say the primary is *concentrated* at the receiver.

![Figure 3-2 Primary dilution and concentration](image)

- **Primary transfer:**
  
  If both primary dilution and primary concentration happened in one hoarding session for the same data, we know it is currency P-hoarding, in which the primary is transferred from the sender to the receiver.

### 3.2.2 Data Hoarding Modes

There are two channels for a mobile host to hoard data: one is *direct hoarding* from another host via pair-wise communication or docked connection, the other is hoarding from broadcast. Data hoarding implies currency hoarding since currency is always accompanying data. Data hoarding releases currency, therefore changes object semantics.

The following notation is used to specify object status when hoarding an object:

- Before hoarding: Sender (object, original sender semantics)
After hoarding: Sender (object, current sender semantics), Receiver (object, original sender semantics, receiver semantics)

Where sender is the host that a primary or copy is hoarded from; receiver is the host that receives the primary or copy; Sender and receiver might be fixed hosts or mobile hosts. Additionally, broadcast server is the sender for hoarding in broadcast disks. Sender/receiver semantics means the sender/receiver holds the primary or a copy of the object.

In the figure below, $X_P$ is the primary of $X$ and $X_C$ is a copy of $X$. “-” means no primary or copy of data $X$ is available in hosts.

Figure 3-3 Data hoarding

- General hoarding (G-hoarding):
  
  Sender owns the primary of an object, and receiver hoards a copy from the sender.

  After G-hoarding, sender still owns the primary.

Before hoarding: Sender (object, Primary)

After hoarding: Sender (object, Primary), Receiver (object, Primary, Copy)
• Primary hoarding (P-hoarding):

Sender owns the primary of the object, and receiver hoards the primary from the sender. After P-hoarding, the primary is transferred from the sender to the receiver and the one in the sender becomes a copy.

Before hoarding: Sender (object, Primary)

After hoarding: Sender (object, Copy), Receiver (object, Primary, Primary)

P-hoarding can only be done via direct hoarding. It is not allowed to P-hoard data from broadcast.

• Copy hoarding (C-hoarding):

Sender owns only the copy of the object, and receiver hoards a copy from the sender. After C-hoarding, both the sender and the receiver held copies of the object.

Before hoarding: Sender (object, Copy)

After hoarding: Sender (object, Copy), Receiver (object, Copy, Copy)

3.3 Currency and Data Synchronization Modes

Any mobile host or fixed host may issue an update to its local data with an updateable-copy currency. Such an update is called a tentative update. To commit a tentative update, we borrow Deno’s weighted-voting scheme to perform currency synchronization: a mobile host issues an election to corner plurality of currency to commit its update.

Data synchronization is done by anti-entropy sessions. In terms of data synchronization, an anti-entropy session is a unidirectional flow of information specifying elections that have been won and propagating the winner updates. Version vectors are used to compare the differences between the communicating hosts. The state of database will reach consistence eventually via anti-entropy.
3.3.1 Currency Synchronization Modes

The idea of currency synchronization comes from that of quorum in distributed databases. Replication of data is commonly used in distributed databases to increase the availability of services. In most cases, the consistency of the data must be maintained despite node failures and/or network partitions. This can be achieved by requiring that, in order to succeed, read and write operations obtain permission from certain sets of replicas. These sets, called read and write *quorums*, are defined in such a way that any two write quorums as well as any read and write quorums have at least one node in common. Then, no two writes can succeed simultaneously thus excluding the possibility of write conflicts. In addition, if every write performs on all replicas from a quorum, a successful read operation is guaranteed to see at least one most recent replica of the data.

A *currency quorum* for an object consists of a set of replicas whose currency summary is the majority of total fixed currency. An update should be accepted by a currency quorum to get commitment. In this way, no two updates can succeed simultaneously thus avoiding the possibility of conflicts. A primary has the currency greater than 50, which forms a currency quorum, thus have the super power to commit an update.

The definition of a currency quorum may be extended to compose a set of replicas whose currency summary is the *plurality* of total fixed currency. When the majority currency is unavailable but a set of replicas hold a currency summary that is greater than that of all the other sets of replicas, it is said that the plurality of currency form a currency quorum. Such currency quorum has the same power to commit an update. For example, three sets of replicas get currencies as 40, 30 and 30. None of them can reach a majority.
since the total is already 100. Then the set with the plurality currency of 40 gets its update to commit.

**Voting to collect currency to commit updates:**

We assume a replication model in which data do not need to be replicated at all hosts and multiple data can be replicated at the same host. A mobile host hoards data before disconnection and reads local data and/or issues tentative updates. A fixed host may issue tentative updates too. If the host holds the data’s primary currency, a tentative update may get committed at once. Otherwise, a tentative update seeks being accepted globally by all the other hosts in the system. It is necessary to collect the majority or plurality of total currencies to commit the tentative update. This process is done by asynchronous weighted-voting scheme. We borrow it from Deno and extended it to include cases of distributed database and broadcast disks.

An election is issued when a host decides to run to collect currencies to commit its tentative update. Any election may have multiple candidates, which represent logically concurrent tentative updates. Candidates from different elections might exist in the system at the same time. A mobile or fixed host holding a non-zero currency for an object X is a *voter* for elections of X. A voter becomes a candidate if it has a tentative update and has not voted its currency to other candidates. A candidate always votes its currency for itself.

Each voter independently collects votes from other voters and deduces outcomes. Voter \( V_i \) maintains following information for each object:

1) \( V_{i.completed} \): the number of elections completed locally.

2) \( V_{i[j]} \): index of candidate voted for by \( V_j \) in \( V_i \’s \) current election, or unknown, i.e. \( V_i \) has not yet seen a vote from \( V_j \).
3) $V_i.currency[j]$: currency voted by $V_j$ in $V_i$'s current election, or unknown.

A candidate $C_j$ wins $V_i$'s current election when:

1) $\text{votes}(V_i, j) > 50$, or // $C_j$ gathers majority of voting currencies

2) $\forall k \neq j$: $\text{votes}(V_i, k) + \text{uncommitted}(V_i) < \text{votes}(V_i, j)$, // $C_j$ gathers plurality of currencies or: $(\text{votes}(V_i, k) + \text{uncommitted}(V_i) = \text{votes}(V_i, j))$ and $(j < k)$  // break a tie

Where $\text{votes}(V_i, j)$ is the summary of currencies known to $V_i$ voting for $C_j$, and $\text{uncommitted}(V_i)$ is the summary of currencies unknown to $V_i$ for $C_j$.

When $C_j$ is awarded as the winner of $V_i$’s currency election, the tentative update issued by $C_j$ gets committed at $V_i$. Currency synchronization for this update finished. The commitment will be propagated to other votes via data synchronization.

3.3.2 Data Synchronization Modes

When a candidate update wins an election, it is committed at the host that awards the election. Then the commitment information is propagated via anti-entropy sessions to other hosts eventually. Version vectors are used to decide which host has the up-to-date version in an anti-entropy session.

**Version vectors.** Every data item has a tentative local version number and a finalized/committed global one. Any tentative update will cause the tentative version number increase, while only committed updates lead to finalized version number increment. All the finalized version numbers of all the shared data items form the version vectors. For a host, if data item X has not been hoarded, its version vector is undefined. Otherwise, version vectors compactly represent the set of updates known to a host.

**Anti-entropy.** The goal of anti-entropy in terms of data synchronization is for two replicas to bring each other up-to-date. Version vectors are used in anti-entropy session to enable a host to correctly determine which updates are unknown to the host by
comparing the version number of a data item to that of the host’s anti-entropy partner. The one has a larger version number will bring the one with a smaller version number up-to-date.

When a tentative update is issued, it gets a locally monotonically increasing tentative version number. Upon this update gets committed via currency synchronization, a globally monotonically increasing committed number is assigned. As it propagated via anti-entropy, all other replicas know the committed update and increase their committed version numbers.

A host can choose its anti-entropy partner at random or based on other knowledge, such as network characteristics or hint of location of primary. The pair-wise anti-entropy protocol is designed to be uni-directional. One host brings another one up-to-date by propagating those updates not yet known to it. The protocol relies on the theory of epidemics to ensure that committed updates eventually propagate to all other replicas [8]. Therefore, the database states will eventually reach consistent. In this way, data synchronization is done.

**An update’s life cycle.** From the above description about data/currency synchronization, we can see that the life cycle of an update that survives an election is following:

- A host issues a tentative update
- The host becomes a candidate in an election
- Updates and votes are propagated in a pair-wise fashion
- Updates gather votes as they pass through hosts
- An update commits when it gathers majority or plurality of votes (other tentative updates are aborted)
- Update commitment information is propagated to reach all hosts eventually via anti-entropy sessions.

### 3.4 Hoarding and Synchronization Algorithms

#### 3.4.1 Definition of Work Sets

We first give definitions of work sets that will be used in pair-wise connection sessions.

The Hoarded Set (HS) for a host consists of primaries and copies hoarded, for which currencies are defined, i.e. greater than or equal to 0. The Data Sync Set (DSS) includes primaries and copies with updates committed. For read-only copies with zero currencies, committed updates made elsewhere should bring the read-only copies up-to-date. The Currency Sync Set (CSS) is composed of primaries and copies with tentative updates in election, which hold non-zero currencies. The Request Set (RS) consists of data requested by applications in a mobile host but not hoarded yet, i.e. currencies are undefined. The Currency Redistribution Set (CRS) includes primaries and copies that need adjust their currencies via currency exchange referring to metadata, which will be detailed in chapter 4. The Global Work Set (GWS) is a logical set, which is the union of HS and RS of all hosts in the database system.

For a host, there are following relations among its work sets:

\[
\text{DSS} \subseteq \text{HS} \subseteq \text{GWS} \\
\text{CSS} \subseteq \text{HS} \subseteq \text{GWS} \\
\text{CRS} \subseteq \text{HS} \subseteq \text{GWS}
\]
\[ HS \cap RS = \emptyset \]

DSS, CSS and CRS may or may not have pair-wise intersections.

The sets for a host are shown as following:

![Global Work Set Diagram]

Figure 3-4 Relation of work sets

The above definitions are used in hoarding and synchronization algorithms. Upon each pair-wise connection, there are four tasks to do in order:

1) Data synchronization: propagate committed updates in DSS via anti-entropy.

2) Currency synchronization: collect voting currencies for tentative updates in CSS.

3) Data hoarding and accompanied currency hoarding: hoard requested data in RS, and allocate currency.

4) Currency redistribution: redistribute currencies for data in CRS between peers.

Tasks 3 and 4 need to refer metadata, which is the topic of chapter 4. Currency redistribution is focused in chapter 5.

For broadcast, only tasks 1 and 3 are performed. Moreover, the hoarding currency is 0 in task 3.
3.4.2 Hoarding Algorithms

Data hoarding is always accompanied by currency hoarding (although in broadcast, the hoarding currency is 0), but not vice versa. The amount of currency dedicates semantics of data. Data semantics may be changed via currency exchange.

3.4.2.1 Direct Hoarding via Pair-wise Communication

Assuming that host S is the sender and host R is the receiver in the hoarding session. The algorithm that R hoards data and currency from S is following:

For each data item X in R.RS //R.RS: request set of R
If X ∈ S.HS (hoarded set of S) and R is authorized to hoard X
S sends data X to R //R hoards data from S
R.X.tentativeVersionNumber = 0
R.X.committedVersionNumber = S.X.committedVersionNumber

//currency hoarding
R consults S to decide initially allocated currency C
S.X.currency -= C //decrease currency from sender
R.X.currency += C //increase the same amount of currency

//copy election and voting information
R.X.completed = S.X.completed //completed # of elections
R.X.V[r][i] = S.X.V[s][i] //R votes for the same candidate as S, or unknown, i.e. not vote yet. r and s are index of R and S, i is the current election that is active.

R.RS = R.RS - {X} //remove X from request set of R
R.HS = R.HS ∪ {X} //add X to hoarded set of R

In the above algorithm, the same amount of currency is decreased from the sender and increased at the receiver to maintain the fixed total currency per object. The receiver votes for the same candidate as the sender if sender voted, or unknown if the sender has not done so. The number of completed elections is actually the committed version number of the data.
3.4.2.2 Hoarding via Broadcast

Only committed versions of data are allowed to broadcast. Any host hoards data from broadcast can only get read-only copies, i.e. currency of data hoarded from air is always 0.

Assuming that host R is the receiver in the hoarding session, R hoards data and currency from broadcast server B using following algorithm:

While R is listening to broadcast
  If the current broadcasted data item \( X \in R.RS \) (request set of R)
    R grabs data \( X \) from the air //R hoards data from broadcast
    R.X.tentativeVersionNumber = 0
    R.X.committedVersionNumber = B.X.committedVersionNumber
    //currency hoarding
    R.X.currency = 0 //R is assigned a read-only copy currency
  //copy version information
  R.X.completed = B.X.completed //the number of completed elections is actually the committed version number of data X

R.RS = R.RS - \{X\} //remove \( X \) from request set of R
R.HS = R.HS \cup \{X\} //add \( X \) to hoarded set of R

There is no need to copy election and voting information in hoarding via broadcast, because data copies hoarded from broadcast are read-only copies with zero currencies, which have no right to vote. Read-only copies may change their semantics to updateable copies or even primaries via currency exchanges detailed in chapter 5.

3.4.3 Synchronization Algorithms

Data hoarding is always accompanied with currency hoarding, while data synchronization is separated from currency synchronization. Voting scheme is used to perform currency synchronization, and data synchronization is done via anti-entropy after currency synchronization is finished. So we talk about currency synchronization first.
3.4.3.1 Currency Synchronization

Currency synchronization is performed when two hosts, say sender S and receiver R, get connected. Synchronized currencies flow from S to R and collect more amount if there are compatible currencies in R. It is said currencies are compatible if they vote for the same candidate in current election or at least one of them is unknown. For a data item X, if both S and R know the most recent committed version number of X, and S has voted for current election of X while R has not, then R votes for the same candidate as S.

Data synchronization will bring previously committed information from S to R, so we need not worry about that S and R have different most recent committed versions.

For each data item $X \in S.CS$ \textit{//} $X$ is seeking currency synchronization

\[ j = \text{the index of the candidate that S votes for in the currency election of X} \]

If $X \in R.HS$ \textit{//} R has a replica of X

If $R.X.V_i[r]$ is unknown \textit{//} R has not vote for current election

\[ R.X.V_i[r] = j \]

Get the vote information of other hosts unknown to R from S

Update $R.X.Votes(V_r,k) = \sum_{i=1}^{n} R.X._i.currency[i]$

\text{s.t. } R.X.V_i[i] == k, \text{ for each } k = 1..n

Update $R.X.V_r.uncommitted$

\[ l = \max \{R.X.Votes(V_r,k)\}, \text{ where } k = 1..n \]

If $R.X.Votes(V_r, l) > 50$ \textit{//} majority of currencies

Award $l$ as election winner of $X$

Else for each other candidate $t$ \textit{//} find plurality of currencies

If $R.X.Votes(V_r, l) > R.X.Votes(V_r, t) + R.X.V_r.uncommitted$

Award $l$ as election winner of $X$

There are following work to do when $l$ is awarded as election winner of $X$:

\[ R.X.committedVersionNumber ++ \]

\[ R.X.committed[R.X.committedVersionNumber] = l \]

\[ R.X.CS = R.X.CS - \{X\} \quad \text{// remove } X \text{ from currency sync set since} \]

\[ \text{currency sync of } X \text{ is done} \]

\[ R.X.DS = R.X.DS \cup \{X\} \quad \text{// add } X \text{ to data sync set} \]
Currency synchronization cannot be performed via broadcast, because there is no effective currency flow in broadcast (only zero currencies in broadcast).

3.4.3.2 Data Synchronization

Data synchronization is performed via anti-entropy in pair-wise connection or broadcast. Committed updates are propagated instead of committed database contents. The reason is that the amount of data propagated during data synchronization is proportional to the update activity at the replicas instead of being dependent on the overall size of the data being replicated. Thus, when the database size is much larger than the database updates, the bandwidth required for the execution of the protocol is reduced. Furthermore, the propagation of update operations avoids any ambiguity introduced by the creation and deletion of replicated objects [23]. Protocols based on the exchange of deltas or differences in data values require additional mechanisms to correctly handle this ambiguity because the existence of a value at one replica and the lack thereof at another cannot correctly identify whether the value is new or it has been deleted.

**Data synchronization via anti-entropy by pair-wise connection**

Assuming sender is S and receiver is R.

For each data item $X \in S.DSS$ // X is known by S as committed

If $X \in R.HS$ and

$R.X.committedVersionNumber < S.X.committedVersionNumber$

$R$ performs each committed update unknown to it

$R.X.committedVersionNumber = S.X.committedVersionNumber$

$R.X.CSS = R.X.CSS - \{X\}$ //election winner got awarded elsewhere

**Data synchronization via anti-entropy by broadcast**

Assuming the broadcast server is B and the receiver is R.

While R is listening to broadcast

If the current broadcasted data item $X \in R.HS$

$R$ grabs data $X$ from the air
If $R.X.committedVersionNumber < B.X.committedVersionNumber$
$R$ performs each committed update unknown to it
$R.X.committedVersionNumber = B.X.committedVersionNumber$
If $X \in R.X.CSS$
$R.X.CSS = R.X.CSS \setminus \{X\}$
//election winner got awarded elsewhere.

Data synchronization by broadcast will significantly speed up the process toward database consistency in that all listeners may get the commitment information at the same time and need not wait for the lazy propagation.
CHAPTER 4
METADATA MANAGEMENT FOR MOBILE DATABASES

Recent advances in wireless networking technologies and the growing success of mobile computing devices are enabling new issues that are challenging to mobile database designers. Mobile hosts have to deal with planned or unexpected disconnections when they mobile; they discover other hosts in ad-hoc manner; they are likely to have scarce resources such as low battery life, slow processor speed and limited memory; their applications are required to react to frequent changes in the environment such as new location, high variability of network bandwidth; their data interests are changing from time to time and from location to location; even data semantics in mobile hosts are varying according to data access patterns, connection duration and disconnection frequencies, etc. All of these require a dynamic database design and reconfigure scheme.

In the past decade, there are some technologies like middleware technologies have greatly enhanced the design and implementation of distributed applications [10]. They successfully hide away many requirements introduced by distribution such as heterogeneity, fault tolerance, resource sharing, and etc. But these technologies have been designed for static distributed systems built with fixed networks. They are not suitable for the dynamic characteristics of mobile environment. For example, static distributed systems assume high bandwidth connection and constant availability. While in mobile environments, low bandwidth and disconnection are normal rather than unexpected. Mobile databases have to be aware of and adapt to the varying contexts such as fluctuating network bandwidth, access patterns, decreasing battery power, location
changes and so on. In order to dynamic design and reconfigure the mobile databases, it is necessary to use metadata to record the changing context and adjust the data distribution.

Metadata is the activity styles and constraints data of mobile hosts. It consists of connectivity, availability, data access patterns and user-defined constraints. The metadata of a mobile host reflects its priority to hoard data in the network and may be mapped to some amount of currency.

The roles of metadata include:

1) Identify data user cares about (Domain): What data interests me?
2) Specify relative worth of data (Utility): What is its relative worth?
3) Indicate power of mobile host (Priority): What’s my activity pattern?

In this chapter, we first give the general categories of metadata in the mobile databases. Then we focus on a subset of metadata and talk about management algorithms.

### 4.1 Categories of Metadata in Mobile Databases

Generally, metadata in mobile databases is composed of data of connectivity, availability, data access patterns and user-defined constraints.

#### 4.1.1 Connectivity

In mobile databases, connectivity of a mobile host is the ability to connect to or communicate with another mobile host or fixed host. There are several aspects related to connectivity:

1) Connection status: strong connected, weak connected or disconnected?

2) Interconnectivity: Is a mobile host connecting with another mobile host or a fixed host? In the past time period, how many hosts did this mobile hosts connect with?

3) Bandwidth: the capacity for data transfer between the two hosts in their communication.
4) Duration: the time during which connection is available. Connection duration in a time period is the summary of connection time.

5) Discontinuity: disconnection frequencies, and periods between connect durations. This parameter represents the lack of connection of mobile hosts. For example, in a one-hour period, two mobile hosts both have 30 minutes connection time, but the one that has 5 disconnections has different connectivity from the other has no disconnection.

6) Time pattern: statistics data about when a mobile is most likely to connect. For example, does it usually connect in weekdays or weekends, am or pm?

### 4.1.2 Availability

Availability shows us what kinds of data are available in a mobile host, how fast and where it is, and the robustness.

1) Data semantics: according to the semantics represented by currencies, there are four levels of data semantics:
   - Primary: the replica has a primary currency.
   - Updateable copy: the replica has an updateable copy currency.
   - Read-only copy: the replica has a read-only copy currency.
   - Unavailability: the amount of currency is undefined at a mobile host, which means no such data is currently hoarded in.

2) Delay: there are two kinds of delays: one is query delay, which is from the time that an application raises a query to the time that it receives reply. The other is update commit delay, which is from the time an application issues a tentative update to the time when the application gets to know the commitment of its update.

3) Locality: where is a mobile host? Is it in the center, edge, or periphery of a particular region? We may use the triangulation teemingness to get coarse location data. Soon every device in the universe will have a GPS, so it is even possible to know the precise location of a mobile host. What a user cares about might changes radically depending on where he or his possessions are.

4) Reliable and failure recovery: is this mobile reliable? Whether and how fast is it able to recovery from failures?

5) Ability: how much is a mobile host able to do? For example, how much is its memory and how long is the battery life?
4.1.3 Access

Access represents data interests and satisfactory with the system from the point of view of applications or clients of a mobile host.

1). Hot data: who is the hottest? It may be evaluated referring to the read and write frequencies.

2). Tentative update frequency: the number of tentative updates issued by a mobile host.

3). Update commit rate: the number of tentative updates having got committed in a time period.

4). Update commit delay: the mean time of the commit delay for a mobile host.

5). Query satisfactory: what is the rate of reply to request?

4.1.4 User-defined Constraints

Besides the metadata collected from runtime, users may specify some parameters that may affect the database distribution. For example, a user may specify his patience of waiting for a query reply. Another example is a team leader may have more priority in data semantics than his team members when they travel out to support their clients.

4.2 Metadata Collection in Mobile Databases

4.2.1 Focused Subset of Metadata

As described in the previous section, there are quite a number of categories of metadata in mobile databases. It is impractical for us to maintain all of them at this time since there is much work to do and it is out of the scope of this thesis. Therefore, we just focus on a subset of the categories and choose those of them that are good enough to show the benefits of metadata and can be maintained with low overhead. Since the cost is low, it is more efficient to refer to the metadata to dynamically design and reconfigure mobile databases.

The focused subset of metadata is given below:
1) Connection duration

2) Disconnection frequency

3) Access statistics: query and tentative update frequencies

4) Commit rate, and

5) Commit delay.

Here the connection status, duration and disconnection frequencies are metadata of mobile hosts, and the others are metadata of data items, which needs to be maintained for each data item in a mobile host.

The chosen subset of metadata has significant impact to behaviors of mobile databases. Due to the weak and intermittent wireless connection, the cost of communication bandwidth and frequent disconnection, static data and currencies assignment will hurt the performance of mobile databases. For example, if a mobile host changes its connection statistics from strong connection, long duration to weak connection, short duration, should it have the same currency as before? No. It is better to give some of its currency to other mobile hosts that have better connection so they have more power to access and commit data, hence improving the database performance. The metadata of connection status, connection duration and disconnection frequency give the connection statistics of a mobile host. It is important to collect such metadata to reconfigure mobile databases.

The metadata of access statistics and commit delay plays an important role in database reconfiguration. The hotness of data items is pointed out by access statistics. If the access is becoming more frequent than before, it means the data here is becoming hotter and needed more heavily by applications. The currency of data should flow to the
place where the data are accessed more frequently. Commit delay and commit rate may be used to indicate the satisfaction of applications to the services of mobile databases. A mobile host with high priority needs to commit more updates as much as rapid. Such mobile host should get more currency, i.e. the mobile databases are reconfigured.

4.2.2 Metadata Collection

To collect metadata, a monitoring scheme is needed to record what happens in a mobile host. Sliding window scheme is used to implement incremental metadata collection.

A *sliding window* is a time frame with a predefined time interval, and the end point of the frame being the current real time. Window width is the time duration of the sliding window. It might be more precise to use different width of time sliding windows to different metadata since they may have different time characteristics. While for convenience and simplicity, we use the same constant width of sliding window for all metadata involved.

*Events* in a sliding window include connection, disconnection, read, tentative updates and update commitment. Such events are logged in the time order that they occur. Since the capacity of a mobile host is limited, we cannot afford to storage them all from the very beginning. So the metadata collection algorithm is designed to log only the events in the time range of a sliding window. The log is called *bounded window log*.

Sliding windows consume events as well as store them. When an event occurs, if the size of the sliding window has not reach its limit, the event is logged and its effect is reflected to the metadata. If the sliding window grows to its upper limit, an old event is retired and its effect is eliminated from the metadata when the new event is added. In this
way, the metadata always remember the effects of events in the most recent bounded history. Figure 4-1 gives an example of sliding windows for a mobile host.

As shown in Figure 4-1, the x-axis represents time. The two dashed rectangles are marked as sliding windows $w_i$ and $w_k$. $w_i$ is from $t_0$ to $t_{10}$ and $w_k$ is from $t_5$ to $t_{15}$. The events are recorded in these two sliding windows as: connect ($t_1$), disconnect ($t_3$), connect ($t_6$), read X ($t_8$), read Y ($t_9$), tentative update X ($t_{11}$), commit update X ($t_{12}$), read X ($t_{13}$), disconnect ($t_{14}$). In this figure, G-hoarding Y ($t_2$), request X ($t_4$) and C-hoarding X ($t_7$) will not be logged as metadata of the mobile host. They are put there just to show when they occur.

The events happened are recorded in a log. Log length is the time duration from the first event to the last one in the log. The algorithm to collect metadata at a mobile host is given as following:

```
While (true)
  If an event e comes at time t
    // Log the event
    add e to the log
    switch (e)
    case connect:
      save connect status and connect start time
    case disconnect:
      save connect status and connect end time
      increment duration by the real connection time
```
increment disconnect frequency by 1

case read:
    increment read frequency of the accessed data item

case tentative_update:
    increment tentative update frequency of the updated data item
    save the issue time of the tentative update

case committed-update:
    increment commit update frequency of the updated data item
    save the commit time of the tentative update
    calculate commit delay of this update
    calculate the mean commit delay
    calculate commit rate

If log length is greater than the length of a sliding window
    \( t_{\text{start1}} \) = the start time of the previous window
    \( t_{\text{start2}} \) = the start time of the current window

For an event \( e' \) happened at time \( t' \in [t_{\text{start1}}, t_{\text{start2}}) \)
    // Eliminate effects of the event
    switch (\( e' \))
        case connect:
            \( t_{\text{disconnect}} \) = the time of disconnection for \( e' \)
            if \( t_{\text{disconnect}} \in [t_{\text{start1}}, t_{\text{start2}}) \)
                decrement duration by \( (t_{\text{disconnect}} - t') \)
                delete the disconnection event from the log
                decrement disconnect frequency by 1
            else
                decrement duration by \( (t_{\text{start2}} - t') \)
        case disconnect:
            decrement duration by \( (t' - t_{\text{start1}}) \)
            decrement disconnect frequency by 1
        case read:
            decrement read frequency of the data item by 1
        case tentative_update:
            decrement tentative update frequency by 1
        case committed_update:
            decrement commit update frequency by 1
            decrement commit delay
            calculate the mean commit delay
            calculate commit rate
        end switch

Delete \( e' \) from the log

In the above algorithm, the mean commit delay for each data item is maintained
as the total commit delay divided by the commit update frequency in the current time
sliding window. The metadata that is used in currency redistribution includes connection
duration, disconnect frequency, read frequency, tentative update frequency, update commit rate and the mean commit delay.

It is clear from the algorithm that the overhead to collect metadata for a mobile host is trivial. The main overhead is the I/O cost to store the event and update metadata when an event occurs. The storage of the metadata and events needs only a couple of disk pages that is very small comparing to the capacity of the mobile hosts. Other overhead is just some CPU cycles. There is no message cost since it happens totally locally. Therefore, the overhead to collect metadata for mobile hosts is very small and the potential benefits may well worth the effort.

So far we have talked about local metadata for mobile hosts. Logically, global metadata is the union of all local metadata and it is decentralized. To form global metadata, local metadata may be piggybacked with anti-entropy messages to get to be known by all the hosts in the system. For ad-hoc databases, it is performed in a decentralized manner to collect metadata and redistribute currency via pair-wise communication. While in distributed databases, global metadata is more powerful to do reallocation in the point of the view of the whole system.

For a fixed host in the distributed network, the focused subset of metadata mainly includes access statistics, the rate and delay to commit updates. The algorithms to get such information are similar to those for mobile hosts. Such local metadata is maintained for each data item at each fixed host and the union of local metadata is the global metadata. The union operation may be performed explicitly at request of a fixed host, or local metadata may be piggybacked with anti-entropy messages to propagate to other hosts.
Now with the metadata in hand, it is ready to refer it to redistribute currency. It is addressed in the next chapter.
CHAPTER 5
DYNAMIC CURRENCY ALLOCATION AND REDISTRIBUTION

As stated in previous chapters, traditional distributed databases are static in that data allocation is determined and will not be changed. But the dynamic characteristics of mobile environment require that mobile databases are designed and reconfigured in a dynamic manner. The varying characteristics are recorded as metadata. In the mobile databases, the semantics of a data replica is indicated by the amount of currency. Furthermore, the semantics may be changed to adapt to the environment characteristics and database accesses. Metadata is referred to determine the data semantics in that the amount of currency is redistributed and thus data replicas are reallocated.

In the previous chapter, metadata is classified and the collection algorithms are given. This chapter is focused on how to do currency redistribution referring to metadata.

5.1 A Target of Currency Allocation

There are a large number of categories of metadata in mobile databases as described in section 4.1. This thesis is focused on a subset of them: connection duration, disconnection frequency, statistical data of accesses, update commit delay and commit rate.

To allocate currency while taking advantages of metadata, a policy of assigning weights to metadata and mapping it to currency is needed. All the database members have to agree to the weight assignment policy to metadata. Intuitively, positive weights should be given to the metadata of connection duration, read frequency and tentative update frequency. It is obvious that a mobile host with longer connection duration, more
frequent read/write accesses and higher commit rate plays a more important role and should get more currency. While the metadata of disconnection frequency and commit delay should be assigned negative weights since they reveal the inability of a mobile host.

These numbers of weights may be customized according to the expected performance and requirements of the system.

The weight of a replica is the summary of production of metadata weight and the value of metadata, which is composed of two parts: one comes from metadata of the mobile host i.e. connection duration and disconnection frequency, the other consists of weights of metadata of the replica, i.e. read frequency, tentative update frequency, commit update frequency, commit rate, and mean commit delay.

Weight of a replica $X_1$ is defined as $W(X_1) = \sum_{m \in M} m \cdot w(m)$, where $M$ is the union of sets of metadata for the replica and metadata for the mobile host where the replica resides, and $w(m)$ is the weight of metadata $m$.

The mapping of currency to weight is in direct proportion: a replica with more weight will be given more currency, i.e. for two replicas $X_1$ and $X_2$, their total currency is $C$, and they have weights $W_1$ and $W_2$, respectively. Then they should get currencies $C_1$ and $C_2$, respectively, calculated as following:

$$C_1 = C \cdot W_1 / (W_1 + W_2)$$

$$C_2 = C \cdot W_2 / (W_1 + W_2)$$

Metadata is collected only after a mobile host has hoarded a replica. It is not available when the data is first hoarded. In this case, expected metadata or user profiles should be given as the initial metadata.
5.2 Dynamic Currency Redistribution Referring to Metadata

Metadata records access statistics of databases and connection statistics of mobile hosts, i.e. which data are hot and which sites are hot. Currencies flow in the system dynamically and adapt to the use of databases. They are always going toward the place where they are needed most: hot sites and hot replica. But in order to be correct and efficient, it should be careful to exchange currencies among replicas in mobile and fixed hosts.

5.2.1 Correctness Requirements of Currency Redistribution

In the weighted voting scheme, the most important property is that the total currency of an object is fixed for any election. In currency exchanges, this property should be maintained carefully, otherwise if the amount of total currency is not fixed any more, the correctness of the scheme will be hurt: there is no way to determine majority or plurality hence no way to commit updates.

The correctness requirements of currency exchange consist of following two items:

1) *No more*: Any amount of currencies of any replica should not be used more than once in any election.

2) *No less*: Any amount of currencies of any replica is available to every election, i.e. should be used at least once.

From the above two requirements, it is implied that any amount of currencies of any replica will be used *exactly once* in every election. To satisfy the requirements, the currency exchange policies should be designed carefully to select the elections that the exchanged currencies take effective.
5.2.2 Currency Redistribution

In currency redistribution, there are two things to determine: one is how much currency each involved host should get, the other is when or from which election the redistributed currency should take effective? They will be addressed respectively in the following three scenarios:

**Between two weak connected mobile hosts.** In this case, mobile hosts communicate in pair-wise manner. The amounts of currencies are directly proportional to the weights of replicas they have. Assuming both mobile host $M_1$ and $M_2$ have replicas of data item $X$, and hoard currencies $C_1$ and $C_2$ respectively. The weights calculated from metadata are $W_1$ and $W_2$, respectively. Then the target amounts of currencies of the two mobile hosts should be the following:

Currency for the replica in $M_1$ is $C'_1 = (C_1 + C_2) * W_1 / (W_1 + W_2)$, and

Currency for the replica in $M_2$ is $C'_2 = (C_1 + C_2) * W_2 / (W_1 + W_2)$.

So the transferred currency between these two mobile hosts is as following:

$\Delta C = | C'_1 - C'_2 | = (C_1 + C_2) * | W_1 - W_2 | / (W_1 + W_2)$.

These two mobile hosts $M_1$ (with most recent election $e_1$ on data $X$) and $M_2$ (with most recent election $e_2$ on data $X$) would like to exchange (say, from $M_1$ to $M_2$) currency $\Delta C$ of replicas of $X$ in their pair-wise communication. The currency synchronization performed prior to the currency exchange will bring the replica with lower commit version number up to the replica with the higher commit number.

The currency exchange is effective in election $e$, which is defined as following:

1) If $e_1 < e_2$, then $e = e_2$. $M_1$ has not voted for election $e$ ($e$ is unknown in $M_1$), so $\Delta C$ can be used in $e$ of $M_2$. 
2) If \( e_1 = e_2 \):
   - If both \( M_1 \) and \( M_2 \) voted for the same candidate or neither of them voted, then \( e = e_2 \).
   - If \( M_1 \) voted but \( M_2 \) did not: if \( M_2 \) will vote for the same candidate as \( M_1 \), then \( e = e_2 \), otherwise, \( e = e_2 + 1 \).
   - If \( M_2 \) voted but \( M_1 \) did not, then \( M_2 \) will vote \( \Delta C \) for the same candidate as the one \( M_2 \) voted \( C_2 \) for, i.e. \( e = e_2 \).
   - If both \( M_1 \) and \( M_2 \) voted but for the different candidates, then \( e = e_2 + 1 \).

3) If \( e_1 > e_2 \):
   - If \( M_1 \) voted, then \( e = e_1 + 1 \). Since \( \Delta C \) has been used by \( M_1 \) in elections \( e_2, e_2 + 1, \ldots, e_1 \), it can only be used in \( M_2 \)’s future voting from \( (e_1 + 1) \).
   - If \( M_1 \) has not voted for \( e_1 \), then \( e = e_1 \).

**Among fixed hosts and/or strong connected mobile hosts.** Assuming replicas of data item \( X \) are hoarded by fixed hosts and/or strong connected mobile hosts numbered from 1 to \( k \), their current weights are \( W_1, \ldots, W_i, \ldots, W_k \) and their currencies before redistribution are \( C_1, \ldots, C_i, \ldots, C_k \). The amount of currency a replica should get after distribution is directly proportional to its weight:

\[
C_i' = \left( \sum_{j=1}^{k} C_j \right) * W_i / \left( \sum_{j=1}^{k} W_j \right)
\]

The currency redistribution is performed via distributed transactions. Due to the good network communication of fixed hosts and/or strong connected mobile hosts, it can be seen that all of them have the same current election \( e' \). For a host \( i \) who hoards \( \Delta C = \)
(C_i' \text{ - } C_i) \text{ from other hosts, the currency exchange is effective in election } e, \text{ which is defined as following:}

1) If \( \Delta C \) has not been voted or voted for the same candidate as the host \( i \) voted, then \( e = e' \).

2) If \( \Delta C \) has been voted but the host \( i \) didn’t vote: if the host \( i \) will vote for the same candidate, then \( e = e' \); otherwise, \( e = e' + 1 \).

3) If \( \Delta C \) has been voted and the host \( i \) voted but for different candidates, then \( e = e' + 1 \).

It is possible that \( \Delta C \) comes from more than one host, i.e. \( \Delta C = \Delta C_1 + \Delta C_2 + \ldots + \Delta C_m \). Then the host \( i \) hoards \( \Delta C_1, \Delta C_2, \ldots \Delta C_m \) one by one, using the above algorithm to get effective elections for each of hoarded currencies.

**Between a weak connected mobile host and a fixed host.** In this case, the currency exchange may involve one or more neighbors of the fixed host. Considering the scenario that the currency in the fixed host is not enough to meet the requirement of the mobile host, the fixed host may have to borrow some currency from its neighbors. Or the mobile host may want to return more currency to the fixed network and the currency should be distributed to more fixed hosts. These will cause currency shrink or currency swell in the distributed databases. They are performed via distributed transactions and are transparent to the mobile host.

1) A mobile host hoards currency from a fixed host:

It is similar to the case between two weak connected mobile hosts if the fixed host has enough currency to give. Otherwise, the fixed host will borrow currencies from one
or more of its neighbors to cause such neighbors shrink in currencies, as illustrated in the
Figure 5-1.

![Figure 5-1 Currency shrink in fixed hosts to meet the hoarding request of a mobile host](image)

The amount of currency a fixed host shrinks is directly proportional to its weight
of the replica:

\[ \Delta C_i = \Delta C \times \frac{W_i}{\sum_{j=1}^{k} W_j} \]

If \( \Delta C \) comes from only the fixed host, the effective election of the currency
exchange is defined similar to the case between two weak connected mobile hosts. If \( \Delta C \)
comes from more than one fixed host, i.e. \( \Delta C = \Delta C_1 + \Delta C_2 + \ldots + \Delta C_m \), then the mobile
host hoards \( \Delta C_1, \Delta C_2, \ldots, \Delta C_m \) one by one, and get effective elections for each of
hoarded currencies.

2) A mobile host returns currency to a fixed host:

In this case, the mobile host gives some or all of its currency to the fixed host.
Then fixed host then distributed the currency to its neighbors to cause their currencies
swell. There are two steps to go: the first is \( \Delta C \) go to the fixed host, the second is \( \Delta C \) get
distributed.

The amount of currency for a fixed host \( i \) should increase is:

\[ \Delta C_i = \Delta C \times \frac{W_i}{\sum_{j=1}^{k} W_j} \]
In this case, since the fixed hosts in the fixed network have the same election, the effective election of $\Delta C$ is the same for all the involved fixed hosts. It is defined similar to the case among fixed hosts.

### 5.3 Ad-hoc Database Check-out/Check-in Currency

When the mobile hosts are all docked (strong connected), all the mobile and fixed hosts have good communication. The mobile database now becomes a distributed database. When some or all of mobile hosts weak connected, they form an ad-hoc network and their database is an ad-hoc database. The ad-hoc database carries out some or all of currencies and perform database operations in ad-hoc manner. For each individual mobile or fixed host, the currencies are redistributed referring to metadata using the scheme given in section 5.2. Primaries may dilute, concentrate or be transferred in or between ad-hoc database and the distributed database.

The scenario of ad-hoc database checking out currency is given in Figure 5-3.
When the ad-hoc is all docked, the mobile database is actually a distributed database. So traditional distributed database management can be used to run the system, except that metadata collection is performed. When the ad-hoc is half docked or all undocked, the mobile database now consists of ad-hoc database and distributed database. So the database should switch to dynamic, weighted voting scheme to take the advantages of dynamic redesign and reconfigure via dynamic currency redistribution.

The scenario of ad-hoc database checking in currency is shown in Figure 5-4.

![Figure 5-4 Scenario of ad-hoc database checking in currency](image)

5.4 Creation and Retirement of Currency and Data

Assuming that the fixed global work set, once created, will not change. For mobile and fixed hosts, creation of currency and data happens when the hosts hoard a replica. The replica will be allocated some currency while ensuing that the total currency for a data item in the system is always is fixed.

Fixed hosts have large capacity, so they can always store data replicas. Even when they have not used a data replica for a long time, the system just leave its currency
as zero to indicate the infrequent use of data, instead retire the data replica from the fixed host.

The capacity of a mobile host is limited, so it cannot afford the cost to store a replica that it will not need any more, especially when the replica needs a large storage space. After giving all its currency to one of its peers, now the host has a replica with zero currency. To retire the replica, the host has to retire its currency first, i.e. setting its currency to undefined. Then the replica is deleted from the disk of the mobile host. There is some housekeeping to do: remove the replica ID from HS (hoarded Set), delete its metadata, votes and election information, etc.
CHAPTER 6
SIMULATION

To demonstrate the benefits of the protocols given in previous chapters, a simplified simulation has been implemented. It simulates the behaviors of mobile hosts under two cases: dynamic currency redistribution and static currency allocation. The metrics of our focus are commit rate, commit percentage and commit delay. Before qualifying the performance of the protocols, the simulation assumption and settings are described first.

6.1 Simulation Assumption

The simulation has been implemented using Java in a local area network environment, i.e. CISE network. A host is implemented as a process running on a UNIX Solaris machine remotely and UDP data grams are used to communicate among processes. Since UDP is not reliable, in general, we may expect that our data packets could be lost, duplicated or delivered out of order. But in the CISE network, this will not really be a concern.

The focus of the simulation is to show the benefits of dynamic currency redistribution among mobile hosts, so the fixed network is concentrated as one fixed host to avoid currency shrink and currency swell in distributed databases. Four mobile hosts are modeled and each of them has an event creator which generates events to simulate the activities of a real mobile host. The main assumptions are:

1). The database is a single object database, i.e. there is only one object shared among these hosts.
2). Data hoarding is assumed to be performed prior the simulation, i.e. each host has a non-zero currency of the object when the simulation is started.

3). The generated tentative updates are independent so that the abort or commitment of a tentative update does not have any effect on the other tentative or committed updates.

4). Applications read committed updates only. Tentative updates are unavailable before they commits.

5). The mobile hosts are failure free, so we need not worry about currency loss.

The metadata in consideration includes connect duration, disconnect frequency, read and tentative update frequencies, commit rate and commit delay.

6.2 Simulation Model

The simulation is designed to be started by an initiator, which takes a configure file and start hosts according to the arguments as configured. Each host is running remotely and working independently with regard to their own configures after being started. The following figure illustrates the scenario of the initiator and the hosts.

Figure 6-1 Initiator and hosts in the simulation
The simulation of a mobile host consists of a processor and a listener. The algorithm of a processor is following:

Initialize;
While the simulation is not yet finished
    If the message queue is not empty and the host is connected
        Process messages;
    Get next event from event creator;
    Process event;
    Add event to time sliding window;
    If the time sliding window is over size
        Adjust the window;
End while

A listener is implemented as a thread to receive messages from other hosts and put them to a message queue. The message queue is accessed synchronously by the processor and the listener of a host.

There are two types of messages being sent and received: REQUEST messages and REPLY messages. A REQUEST message is composed of committed update table, election, currency and weight of replica. A REPLY message includes committed update table, election and delta currency. The committed update table is used to propagate committed updates among peers in data synchronization. The election is used in voting scheme to collect currencies. The currency and weight of replica are used in currency redistribution.

For a mobile host, if the current event is CONNECT, then the host will send a REQUEST message to the selected peer. The peer will perform data synchronization, currency synchronization and currency reallocation according to their weights and send back a REPLY message to inform the results. Messages in the model of a host are shown in the following figure:
6.3 Simulation Settings

In mobile database applications, access of the database items may vary according to the interests of the mobile applications. So the simulator should mimic the activities of mobile applications and generate different events to cause hot spot transferred among the hosts. To get different frequencies of events, intervals between two consecutive events are configured for each host. Roughly the activity patterns of each host are designed as two sections: the fixed host 0 is configured inactive in the first section but active in the second section; mobile host 4 is configured active in the first section and inactive in the second section; the other mobile hosts are configured inactive in both the first and second sections.

One of the most important parameters is the tentative update ratio. According to the simulation assumption, all the tentative updates are independent. A tentative update should be issued to win an election to get committed. The election performs currency synchronization among replicas and currencies are intended to be redistributed to flow to the hottest place of the system. It is expected that the tentative update ratio have direct
impact to the commit rate and commit delay. Relative to the frequencies of read events, the tentative update frequency ratios are designed from 0.1 to 0.6.

The width of the time sliding window is also important in that a too wide window causes the metadata data changes slowly thus currency flow slowly so may not catch the hot place in time, while a too narrow window is vulnerable to some false frequent events and lead to unnecessary currency flow hence decrease the efficiency of the system.

The scenarios of static currency allocation simulated in this model are uniform currency allocation and primary currency allocation. The former is a case of non-primary scheme and the latter is equivalent to the primary-copy scheme. To compare the performance of the dynamic design to the static design, the scenarios of dynamic currency redistribution design are simulated under the same initial currency allocations as the static cases.

The main simulation parameters and settings are summarized in Table 6-1 as follows.

Table 6-1 Primary simulation parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>windowWidth</td>
<td>6 seconds</td>
<td>Width of time sliding window</td>
</tr>
<tr>
<td>simuTime</td>
<td>80 seconds</td>
<td>The time of simulation running</td>
</tr>
<tr>
<td>simuSection</td>
<td>[0, 40], [40, 80]</td>
<td>Simulation time sections</td>
</tr>
<tr>
<td>updateRatio</td>
<td>0.1, 0.2, 0.3, 0.33, 0.4, 0.5, 0.6</td>
<td>The ratio of tentative updates to read events</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hosts</th>
<th>First section</th>
<th>Second section</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>Internals between two consecutive events (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial currency allocation</td>
<td>Uniform</td>
<td></td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>These are the initially allocated currencies. For static cases, they are not to be changed.</td>
</tr>
<tr>
<td>(total = 100)</td>
<td>Primary</td>
<td></td>
<td>60</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>Intervals between two consecutive events in different time sections</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hosts</th>
<th>Host0</th>
<th>Host1</th>
<th>Host2</th>
<th>Host3</th>
<th>Host4</th>
<th></th>
<th>Internals between two consecutive events (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>First section</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>0.1</td>
<td>Intervals between two consecutive events in different time sections</td>
<td></td>
</tr>
<tr>
<td>Second section</td>
<td>0.1</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>Intervals between two consecutive events in different time sections</td>
<td></td>
</tr>
</tbody>
</table>
The settings show that in the first section, the host 4 is very active and the other hosts are not, while in the second section, the hot point transfers to the host 0 and the host 4 becomes cold. The purpose of such setting is to show the most currency is always intended to flow to the hottest host.

6.4 Simulation Results

The analysis of the simulation results is focused on the commit rate, commit percentage and mean commit delay. The commit rate is defined as the average number of committed updates per time sliding window. The commit percentage is the ratio of committed updates to the total tentative updates generated in the system. The commit delay of a committed update is defined as the difference from the tentative update issue time to the time that a host knows its commitment. Mean commit delay is the mean delay time of committed updates in the system averaged to all the hosts.

The simulation results demonstrate that the dynamic design is more beneficial that the static design. Two scenarios are studied to qualify the performance: one is uniform scenario, i.e. the currencies are initially allocated to each replica uniformly; and the other is primary scenario, in which the primary currency is initially given to one replica. The initially allocated currencies remain unchanged in static cases while dynamically change with weights/metadata of replicas in dynamic cases.

6.4.1 Commit Rate and Commit Percentage

The commit rate and commit percentage of dynamic cases versus static cases of uniform scenario are given in Figure 6-3 and Figure 6-4.
It can be clearly seen that the dynamic design is much more beneficial than the static design except when the update ratio is rather low. Specifically, when the update ratio is 0.1, little benefit of dynamic design is shown over the static design. But when the update ratio is bigger, the benefit becomes significant: the commit rate of the dynamic design increases while that of the static design remains almost unchanged; the commit percentage of the dynamic is almost constant but that of the static design decrease rapidly. The reason is that in the dynamic design, the currency is always flowing to the hottest replicas thus there are more tentative updates to get plural currency and be committed rapidly. While in the static case, the currency allocation is fixed and there is...
no priority for the hottest replica, so the commit rate is almost a constant no matter the number of tentative updates generated.

Figure 6-5 and Figure 6-6 illustrate the commit rate and commit percentage of dynamic cases versus static cases of primary scenario:

We can learn from the above figures that in the primary scenario, the difference of dynamic and static cases is not so significant as the uniform one. The reason is that the primary currency happens to be allocated to one of the hottest replicas so the performance of static design is not too bad since a fixed primary currency makes all the local tentative updates committed. It can also be seen from Figure 6-5 and Figure 6-6 that the dynamic case still beat the static case when the update ratio is higher than 0.3 even one of hot replica is lucky to get primary currency in the static case.
It is reasonable that the benefits of the dynamic design will be more significant, even better than the uniform scenario, if the primary currency is unfortunately given to a cold replica.

### 6.4.2 Mean Commit Delay

Figure 6-7 and Figure 6-8 demonstrate the commit delay of dynamic cases versus static cases in uniform and primary scenarios:

![Figure 6-7 Commit delay of uniform scenario](image)

![Figure 6-8 Commit delay of primary scenario](image)

The above figures compare the dynamic design and the static design for uniform and primary scenarios in terms of how fast they commit updates. It can be observed that in the uniform scenario, the commitment of dynamic design is much faster than that of static design except when the update ratio is rather low. But in the primary scenario, the
benefit is not so significant because the fixed primary currency in static design does
speed the commitment in the primary replica.

The main overhead of the dynamic design is metadata management and currency
redistribution. As stated in chapter 4, the overhead to collect metadata is trivial. The main
overhead related to currency redistribution is message cost, but the information can be
piggybacked with the anti-entropy messages and that is only a few bytes thus save
bandwidth of mobile hosts. Therefore, the overhead of the dynamic design is little.
CHAPTER 7
CONCLUSIONS AND FUTURE WORK

This thesis is focused on flexible protocols for dynamic database design and reconfiguration in mobile environments. Traditional distributed database design is static, while mobile environments require a dynamic design to support mobility. In this thesis, the concept of mobile database is given first, and then a dual data/currency hoarding and synchronization protocol is introduced. Metadata is used to allocate currencies dynamically so that the mobile database is reconfigured such that the replica location, replication and data semantics can be changed dynamically via currency redistribution referring to metadata. The benefits of dynamic design versus static design are shown via the simulation results. Mobile databases consist of a large scope of research and application issues. In this thesis, what we addressed is only a small subset of it. There is a lot of future work to do:

1) This thesis is focused on just a small subset of metadata. In order to get better performance adaptive to the system changes, the full set of metadata should be considered. This requires a more completed metadata collection mechanism.

2) Associate rules to predict metadata to build rule-based adaptive currency redistribution. The metadata in this thesis is collected only after the events happen, i.e. it comes from history of the system. If there is a scheme to predict what a mobile host needs in the future and hoard the data in advance, the performance will be brought up significantly.

3) The concept of broadcast databases is introduced and included in the hoarding and synchronization mechanism. Detailed algorithms and more use of broadcast should be explored. It should be more powerful to use broadcast to propagate update commitment, election winner awards and hint of primary location.

4) Data management plays a crucial role in mobile computing. Decades of experience with query processing, transactions, replication, caching etc provide a solid
base of technology on which to build. But, mobile computing brings challenges in all aspects of data management. Query processing in mobile databases has new characteristics and should be performed in an optimal way. Mobile transaction models will be built based on the traditional transaction models. For example, the operations in mobile databases include hoarding and synchronization as well as read and write.

5). A complete simulation/implementation is needed to show the benefits of dynamic design of mobile databases.
LIST OF REFERENCES


BIOGRAPHICAL SKETCH

Yanli Xia was born in Sanyuan, Shaanxi Province, in China. She received a Bachelor of Science degree in computer science and engineering from Nanjing University of Aeronautics and Astronautics, Nanjing, China, in July 1997. Then she entered the graduate program at Nanjing University of Aeronautics and Astronautics and obtained a Master of Engineering degree in computer science in April 2000.

She enrolled at the University of Florida in August 2000 in the Department of Computer and Information Science and Engineering. She worked as a teaching assistant, and then research assistant for Dr. Abdelsalam (Sumi) Helal. Her future research interests include mobile databases and distributed systems.