A Caching Policy to Support Read-only Transactions in a Mobile Computing Environment

Man Hon Wong Wing Man Leung
Department of Computer Science
The Chinese University of Hong Kong
Shatin, N.T. Hong Kong
email: mhwong@cs.cuhk.hk, wmleung@cs.cuhk.hk

Abstract

Mobile computing is a new and challenging technology which will definitely revolutionize the way we use computers in the next decade. A mobile computing environment consists of stationary supporting servers and a set of mobile computers connecting to the servers by a low-bandwidth wireless network. The main difference between a traditional client-server environment and a mobile computing environment is that the mobility of mobile computers is unrestricted. Therefore, a mobile computer can disconnect from a server and connect to a different server at any time. In addition, a server may not have any information about which mobile computers are currently connecting to it. Hence, the issue of maintaining a consistent cache becomes complicated. In this paper, a caching policy is proposed to maintain cache consistency for mobile computers so that locks are not required for read-only transactions from mobile computers while serializability is guaranteed.

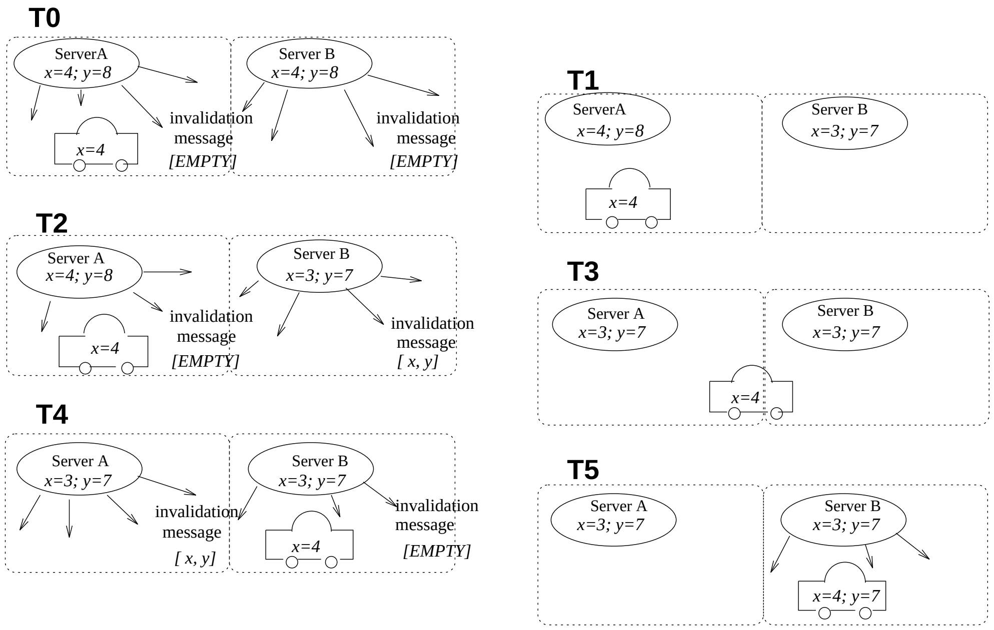
1 Introduction

With the advances of the communication technology, users will soon be able to access databases through wireless networks [9, 6, 11]. In a mobile wireless computing environment, there will be a number of fixed hosts connected by a fixed network and massive mobile computers connect to these fixed hosts through a wireless interface. Most of these mobile computers are low powered desktop or palmtop machines. The area covered by a fixed host is a *cell*. These mobile computers may frequently disconnect from the fixed hosts and cross the boundary to different cells for various reasons. Users of these mobile computers may frequently query databases by invoking a series of operations, generally referred to as a transaction. Serializability is a widely accepted correctness criterion for executions of transactions [8]. This correctness criterion is also adopted in this paper.

In order to reduce the contention on the narrow bandwidth of the wireless channel, usually part of a database is cached in mobile computers [7, 2, 3]. Some caching strategies for mobile computers have been introduced in [3]. The performance of these strategies and the impact of client's disconnection times on these strategies are evaluated. In [11], various static and dynamic data allocation (caching) methods are proposed to optimize the communication cost between a mobile computer and a fixed host that stores the online database. Performance is the main concern of these two papers. Their results are mainly based on the property that communication in a mobile computing environment is expensive and unreliable. The impact of the unrestricted mobility of mobile computers is not addressed. In addition, the notion of transactions and the synchronization issues for executions of transactions are not studied. In an attemp to complement the previous results, a caching policy to support read-only transactions in a mobile computing environment is proposed.

Our goal is to ensure data cached in a mobile computer is consistent in the sense that transactions reading these data are serializable. The issue of maintaining cache consistency for mobile computers is not as simple as that of a client-server based database system. In a client-server based database system, locks are required to be held in the server before data is cached in a client. A server may be required to invalidate outdated data in a client from time to time. By using the combination of locks and invalidation messages, it is easy to keep the data cached in a client consistent. However, in a mobile computing environment, mobile computers may disconnect from servers frequently without notifying the servers. This may due to fail of receiving the transceiver's signal or running out of battery. A server may not even know the existence of every mobile computer. Therefore, delivery of invalidation messages is not guaranteed. In addition, a mobile computer may connect to a different server when it crosses the boundary of a cell. Such a phenomenon further complicates the caching strategy in a mobile computing environment.

Consider a simple cache management policy that mobile computers can request data from servers at any time and servers broadcast invalidation messages periodically to inform mobile computers to drop outdated data. The performance of this approach has been studied in [3]. Figure 1 illustrates a scenario of using this policy that the data cached in a mobile computer can become inconsistent in the sense that transactions reading these data may not be serializable. The main reason for such a phenomenon is that a mobile computer may load data into its cache from different servers. It is well understood that no matter what replica control or commitment protocols are used, it is impossible to update all copies of a data item on different servers simultaneously. Therefore, the data in the cache of a mobile computer may not correspond to part of a database snapshot. Hence, transactions reading these data may not be serializable.



At Time T_0 The values of x and y at servers A and B are the same and a mobile computer is residing at cell A.
At Time T_1 The copies of x and y at server B are updated by a transaction.
At Time T_2 Before A receives the propagation message from B,
server A and B broadcast invalidation messages to their local mobile computers respectively. .
The invalidation messages in these two cells are different.
At Time T_3 The mobile computer crosses the boundary to cell B and server A receives the propagation message from server B for updating x and y.
At Time T_4 After the mobile computer has arrived at cell B, both servers send invalidation messages.
However, the invalidation messages from server B is empty.
At Time T_5 If the mobile computer loads the value of y into its cache, the cache in the mobile computer is inconsistent
in the sense that it does not correspond to a snapshot of the database.

Figure 1: Problem with the Existing Strategy

The most obvious solution to the above problem is to drop the entire cache once a mobile computer crosses the boundary to another cell. However, such a strategy is very inefficient especially when a mobile computer is traveling to and fro the boundary of different cells [12]. In this paper, a new caching strategy is introduced such that the cached data of a mobile computer must correspond to a snapshot of a database. Hence, read-only transactions can read data from the cache without issuing any lock to the servers and at the same time serializability is maintained. The main advantage of our strategy is that the bandwidth can be saved by eliminating the communication required for locking. In addition, the round trip delay time for the locking messages is saved. Furthermore, cache consistency is ensured even when a mobile computer crosses the boundaries of cells without dropping the cached data.

The paper is organized as follows. Section 2 presents the model of the mobile computing environment used in this paper. The version control mechanism, which manages different versions of data items, is described in Section 3. The proposed cache management policy is explained in Section 4. Section 5 discusses some caching issues for partially replicated data items. We conclude our work and suggest future work in Section 6.

2 System Model

In a mobile computing environment, there are two distinct sets of entities: servers and mobile computers. It is assumed that each server and each mobile computer can be identified by a unique identity number. *Servers* are usually powerful stationary computers connected by a reliable network with a high bandwidth. On the other hand, *mobile computers* are portable computing devices (e.g. palmtop or desktop machines). They are usually low power machines equipped with limited amount of memory [16]. Mobile computers are connected to a server through a wireless network. *A wireless network* provides a wireless communication like modulating radio waves or pulsing infrared light between servers and mobile computers [9]. The network between mobile computers and a server is usually unreliable and with a low bandwidth. The area covered by the transceiver's signal of a server is known as a *cell* [2, 1]. Although cells may be overlapped physically, each mobile computer can logically connect to one server at a time [1]. It is assumed that mobile computers cannot communicate directly with each other. Communication between a server and mobile computers is achieved by broadcasting messages in the wireless media. The system enables unrestricted mobility and connectivity of the mobile computers among the cells. Mobile computers may be disconnected from a server for various reasons and cross the boundary of two different cells without notifying the servers.

Users of mobile computers may frequently query information in databases through the servers. It is assumed that the local supporting server is the only source of data for a mobile computer. A database is a collection of data items which may be distributed or replicated among the servers. It is assumed that a data item is the unit of a single access. In practice, a data item may be a record, a relation, a page of memory or even an entire file. Data items are updated by transactions [5]. Each transaction is composed of a series of read and write operations. A transaction which contains write operations is known as an *update transaction* and a transaction with read operations only is known as a *read-only transaction (query)* [8]. The execution of a transaction must be atomic, i.e, a transaction either commits or aborts. If a transaction commits, the effects of the transaction will be incorporated completely to the database. Otherwise, if it aborts, none of the effects will be incorporated into the database. The atomicity of transactions involving multiple servers can be enforced by using a commit protocol [18, 13]. *Serializability* [8], a widely accepted correctness criterion for execution of transactions, is adopted in this paper. Serializability guarantees that an execution of transactions must be equivalent to a serial execution of the transactions. Serializability can be enforced by employing a concurrency control protocol such as the two-phase locking protocol [8] or timestamp ordering protocol [15]. Since a data item may be replicated on several servers, these copies of a data item must appear as a single logical data item to transactions. This property is called *one-copy equivalence* and is enforced by a replica control protocol such as the quorum protocol [10]. By using a quorum protocol, a write operation may not update all the copies of a data item. A replica control protocol together with a concurrency control protocol can enforce one-copy serializability [10], a correctness criterion for replicated database.

After a write operation is executed on a data item , a new version of the data item will be created. If a new version always replaces the old version, then the system is a *monoversion* database system. Otherwise, if it keeps more than one version of the same data item, it is a *multiversion* [14] database system. In this paper, it is assumed that the database is a multiversion database. Thus, each data item x in the database, is represented by a totally ordered sequence of versions, denoted by $x^1, x^2 \cdots x^n$, where the superscripts are the monotonically increasing version numbers. We assume in this paper that the version control protocol proposed in [17] is employed in the servers. The details of the version control mechanism will be discussed in section 3.

3 Version Control Mechanism for Servers

The version control mechanism in [17] is adopted in the servers. By using the version control mechanism, each update transaction, T, is assigned a timestamp $tn(T)$, which is unique and corresponds to the serialization order of the transaction. Therefore, if a locking concurrency control protocol is used, a timestamp is assigned at the end of a transaction when the serialization order is known. In addition, all data items updated by transaction T are stamped with the version number $tn(T)$, i.e., the timestamp of the transaction. A counter called the *visible transaction counter*, *vtnc*, is kept at each server. The visible transaction counter for a server is in fact the lower bound of the timestamp for the next transaction committed in the server. The counter is implemented in this way. When a transaction T executed in a server commits, *vtnc* of the server will be incremented to $tn(T)$, if no transaction T' at the local server will commit with a timestamp $tn(T')$ such that $tn(T') < tn(T)$. All transactions that start on the server after T has committed will be serialized after T and obtain a timestamp greater than $tn(T)$, i.e., greater than $vtnc$. Therefore, no new version can be created with version number less than or equal to $vtnc$ at the server. Note that the $vtnc$ on different servers may be different; hence, a transaction executed on a remote server may commit with a timestamp less than the $vtnc$ of a local server.

By using a quorum protocol, a write operation may not update all the copies of a data item. Therefore, a server may only contain a partial version sequence of each data item. In order to obtain a complete version sequence of each data item, a background propagation mechanism is employed to propagate the updates to every server. A server's local sequences are said to be complete up to the value of a given transaction counter if it contains all versions of the data items with version number less than the counter globally. The extent of completeness of a server's version sequences is represented by a *completeness transaction number counter, ctnc* maintained at each server.

Each server, S_i also maintains two vectors, $V_i[j].vtnc$ and $V_i[j].ctnc$. $V_i[j].vtnc$ indicates S_i's knowledge of the value of $vtnc$ at server S_j; i.e., S_i knows that versions with version numbers less than or equal to $V_i[j].vtnc$ cannot be created at S_j. Similarly, $V_i[j].ctnc$ represents S_i's knowledge of the value of $ctnc$ at server S_j. In addition, each server S_i, maintains a log [19], L_i, which contains the update history of the data items maintained in the system. The log is composed of a sequence of records $\langle id, vn, val\rangle$, where id identifies the data item, vn identifies the version number of the data item and val is the value associated with the version of the data item. Periodically, each server sends propagation messages to all other servers. A propagation message includes a copy of the server's local vectors $V_i[].vtnc$, $V_i[].ctnc$ and a portion of its *log* which contains updates that may not be known by the receiver (records with vn greater than $V_i[j].ctnc$). Upon receiving the propagation message from server S_i, server S_j merges the incoming log with its local log. Then vectors $V_j[i].vtnc$ and $V_j[i].ctnc$ will be updated by taking the pairwise maximum of the local and the received vectors. It has been shown in [17] that no version of a data item can be created with a version number less than $MIN_i(V_j[i].vtnc)$ globally. In addition, the local version sequences at server S_j are completed up to $MIN_i(V_j[i].vtnc)$. Hence server S_j can roll forward its $ctnc$ to $MIN_i(V_j[i].vtnc)$. Readers are encouraged to refer to [17] for the detailed description of the version control mechanism. As a conclusion for this section, we state the two properties of this version control mechanism that are used in our cache management policy.

Property 1 *No transaction in the system will commit with a timestamp less than ctnc of any server, i.e., no version of a data item can be created in the system with a version number less than the ctnc of a server.*

Property 2 *A server S_i contains all versions of data items with version number less than or equal to ctnc.*

4 Cache Management

In this section, a cache management policy is introduced to support read-only transactions in mobile computers. In a mobile computing environment, we have to face additional difficulties that do not exist in a stationary client-server environment. First a mobile computer may cross the boundary and connect to a different cell without notifying the server. By using a quorum protocol for replica control, a write operation may not update all copies of a data item. In addition, it is impossible to update different copies of the same data item at the different servers simultaneously. Therefore, the data in the new and the old servers may correspond to different snapshots of the database. Hence, the consistency in the cache may be destroyed if the invalidation messages and data from the current server are accepted blindly. An example has been shown in Section 1. Secondly, a mobile computer may be frequently disconnected from a server for various reasons, such as failure in receiving the transceiver's signal. Therefore, a mobile computer may fail to receive some invalidation messages from the server. Hence cache consistency could also be destroyed. A trivial solution to these problems is that the entire cache is dropped when the mobile computer crosses the boundary, or missing of a message from the server is detected. However such an approach is very inefficient.

In this section, an efficient protocol which can maintain the consistency of cached data items in mobile computers is proposed. Intuitively, the data items cached in a mobile computer is consistent, if the data correspond to a snapshot of the database. In order to maintain this property, different versions of data item, invalidation messages and data broadcasted to mobile computers are required to be managed in a coordinated manner.

4.1 Cache Consistency

The most crucial property that must be maintained by the proposed caching policy is that the data cached in a mobile computer must be consistent in the sense that transactions reading these data are serializable. Therefore, the data in the cache must be part of a consistent database state. It is generally accepted that executing a transaction can bring a database from one consistent state to another consistent state. Hence, any database state resulted from a serializable execution must be consistent. The simplest method to obtain a consistent database state is to prevent starting of new update transactions and then obtain the database state after all update transactions have been completed. If serializability is enforced, the state obtained in this way must be consistent. In practice, however, it is undesirable to obtain a consistent database state by blocking all update activities. Instead, multiple versions of data items can be kept and a consistent database state can be obtained by selecting a version from each data item properly. Obviously, for each data item, the version that forms the most recent consistent database state must be created by the last update. Therefore, for any timestamp t, the latest versions of data items with version numbers not greater than t form the most recent consistent database state before t, if no transaction will commit with a timestamp less than t. The formal definition for consistent database states is given as follows:

Definition 1 (Consistent Database State) *Suppose* $\{x_1, x_2 \cdots x_n\}$ *is the set of data items in a database. The set of versions* $\{x_1^{j_1}, x_2^{j_2} \cdots x_n^{j_n}\}$ *is the most recent consistent database state at timestamp* t*, denoted* $DBS(t)$*, if for each data item* x_i*, there does not exist and will not exist a version* $x_i^{j'_i}$ *such that* $j_i < j'_i \leq t$*.*

From the definition of consistent database states, it is not difficult to show the following lemma:

Lemma 1 *For any timestamp* t*,* $DBS(t)$ *is unique.*

Proof: Assume the lemma does not hold. Let $\{x_1^{j_1}, x_2^{j_2} \cdots x_n^{j_n}\}$ and $\{x_1^{j'_1}, x_2^{j'_2} \cdots x_n^{j'_n}\}$ are two distinct consistent database states for $DBS(t)$. Then there must exist k such that $1 \leq k \leq n$ and $x_k^{j_k} \neq x_k^{j'_k}$. Since version numbers are totally ordered, without loss of generality, assume that $j_k < j'_k$. Hence, by definition, $\{x_1^{j_1}, x_2^{j_2} \cdots x_n^{j_n}\}$ is not a consistent database state. It leads to contradiction and the lemma follows. ∎

Due to the limitation of storage, usually only a subset of data item is cached in a mobile computer. We assume that for each data item in the cache, only one version of the data item is kept in the cache. We call this set of versions a *version set*. The consistency of a version set is defined as follows:

Definition 2 (Consistent Version Set) *A version set is consistent with timestamp* t*, if the set of versions is a subset of* $DBS(t)$*.*

We say that a cache is consistent with timestamp t, if the set of versions kept in the cache is consistent with timestamp t. In the implemention, a counter will be used to store the timestamp of a cache. Logically, a cache can be represented by a tuple $\left\langle t, x_{i_1}^{j_1}, val_{i_1}, x_{i_2}^{j_2}, val_{i_2} \cdots x_{i_m}^{j_m}, val_{i_m} \right\rangle$, where t is the timestamp of the cache, x_i^j is the identity of the version and val_i is the value assoicated with the version. Intuitively, cache consistency with timestamp t can be maintained, if we can enforce that the version of each data item kept in the cache is the largest possible version with version number less that t. This property can be stated formally as follows:

Lemma 2 *A version set is consistent with timestamp t, if the version for each data item in the version set is the latest version with version number less than t and no transaction will commit with a timestamp less than t.*

Proof: Since version numbers are totally ordered, if no transaction will commit with a timestamp less than t, then for each data item, there exists a unique latest version of the data item that is less than t. Therefore, the consistent version set with timestamp t for a set of data items must be unique. From Lemma 1, $DBS(t)$ is also unique. By the definition of consistent database state, the version set must be a subset of the consistent database state. Hence the lemma follows. ∎

We can observe from the definition of cache consistency that the data in a consistent cache with timestamp t contains the effects of the transactions with timestamps less than t only. It is not difficult to show that any read-only transaction that reads data from a consistent cache with timestamp t can be serialized immediately after all the transactions with timestamps less than t. A similar proof for a multi-version database has been given in [4]. Therefore, if cache consistency can be maintained, any read-only transaction from a mobile computer can read the data in the cache without issuing any lock on the server and serializability is guaranteed.

As new versions of data items are created continuously by update transactions, the cache in mobile computers should always be kept as up-to-date as possible. Updating the cache in a mobile computer may involve three different processes: dropping some old data, reading in new data and advancing the timestamp of the cache. However, these three processes cannot be performed independently. In order to maintain cache consistency, careful coordination is required. The process of advancing timestamps of cache and dropping outdated data from cache must be done in a single atomic step with respect to the transaction in a mobile computer. When a mobile user requests data from servers, the version that can be loaded into the mobile computer depends on the timestamp, t, of the cache. The details are discussed in the following sections.

4.2 Request Data from Servers

The data item required by the user of a mobile computer may change from time to time. Therefore, unused data item will be dropped from the cache and new data item may be loaded into the cache gradually. From Lemma 2, it is not difficult to observe the following lemmas, since set difference and union do not destroy the assumption in Lemma 2:

Lemma 3 *Let V be a consistent version set with timestamp t. Any $V' \subset V$ is also a consistent version set with timestamp t.*

Lemma 4 *Let V and V' be two consistent version sets with the same timestamp t. $V \cup V'$ is also a consistent version set.*

Lemma 3 implies that a mobile computer can drop unused data items at any time without destroying cache consistency. On the other hand, loading a data item into a cache is more complicated. From Lemma 2, the largest possible version of a data item with timestamp not greater than t is a consistent version set with timestamp t. Then by Lemma 4, we can conclude that the cache consistency of a cache with timestamp t will not be destroyed by loading the the lastest version of a data item with version number less than t into the cache. By using this idea, the algorithm for requesting data items from servers is derived.

When a query in a mobile computer requests a data item that is not cached, the mobile computer will send a $REQUEST$ to its local supporting server. A $REQUEST$ is a tuple, $\langle mobile\_id, t, data\_id \rangle$ where $mobile\_id$ identifies the mobile computer, t is equal to the timestamp of the cache and $data\_id$ is the identity number of the required data item. After receiving a $REQUEST$, the algorithm in Figure 2 is executed.

The situations can be classified into three cases:

- $t < old_{vn}$ (the oldest version that is kept in the server)
  Since the version requested by the mobile computer is no longer available in the server, the query needs to be aborted. The query should be re-executed after the timestamp of the cache is advanced.

```
Procedure ReceiveREQUEST
begin
    Suppose a request ⟨mobile_id, t, x⟩ is received.
    If ( old_vn > t )
        Message ← ⟨mobile_id, ABORT⟩ /* Abort the transaction */
        Broadcast Message
    else if (ctnc > t > old_vn )
        val ← x^j.val where ( ∄x^k s.t. j < k < t)
        Message ← ⟨t, x^j, val⟩
        Broadcast Message
    else if ( t > ctnv )
            Block until ctnv ≥ t
end
```

Figure 2: Algorithm for Processing Data Requests at Server

- $old_{vn} < t < ctnc$

  From Property 1 of the version control mechanism, no transaction will commit with timestamp less than *ctnc*. In addition, from Property 2, the server has all the versions from old_{vn} up to *ctnc*. Therefore, x^j, which is the latest version of x with version number less than t is the required version of the mobile computer. The server will then broadcast the message $\langle t,\ x^j,\ val\rangle$ through the wireless network. The semantics of the message is that x^j is the latest version of x with version number not greater than t and val is the value associate with this version. Therefore, all mobile computers with timestamp t can safely load the data item into its cache. Actually, all mobile computers with timestamp t' such that $j < t' < t$ can also load the data item. The details will be discussed in Section 4.4.

- $ctnc < t$

  Since the version sequences in the server is only complete up to *ctnc* It cannot be guaranteed that the latest version of the data item with version number not greater than t is already contained in the server. Hence, the read request will be blocked until the value of *ctnc* is advanced to a value greater than or equal to t.

4.3 Invalidation Report

As new versions of data items are continuously created by update transactions, the timestamp of a cache has to be advanced, so that more updated versions of data items can be loaded into the cache. This section discusses how to advance the timestamp of a cache and at the same time maintain cache consistency by invalidating data items properly. Intuitively, when the timestamp of a consistent cache is advanced from t_1 to t_2, a data item has to be dropped if the version kept in the cache is not the largest possible version that is less than t_2; otherwise the cache consistency will be destroyed. This idea can be summarized in the following lemma:

Lemma 5 *Suppose V is a consistent version set with timestamp t_1. Then $V-U$ is a consistent version set with timestamp t_2 ($t_2 > t_1$), if*

1. *no transaction will commit with a timestamp smaller than t_2, and*
2. $U = \{x_i^j | x_i^j \in V \wedge \exists x_i^k \; s.t. \; t_1 < k \leq t_2\}$.

Proof: Suppose the assumptions in the lemma hold. Assume $V-U$ is not a consistent version set with timestamp t_2. By Definition 2, there exists a version $x_i^j \in V-U$ such that x_i^j is not the largest possible version that is less than t_2. That means there exists or will exist a version x_i^k such that $j < k \leq t_2$. Since no transaction will commit with a timestamp smaller than t_2, x_i^k is already in the system. By definition, $x_i^j \in U$. Therefore, $x_i^j \notin V-U$. It leads to contradiction. The lemma follows. ∎

From Lemma 5, we observe that when a server suggests a mobile computer to advance its timestamp to t, the server has to make sure that no transaction will commit with a timestamp smaller than t. In addition, the server should know all the updates up to t, so that it can instruct the mobile computer to drop the data items properly. Therefore, a server should not suggest a mobile computer to advance its timestamp beyond *ctnc*.

Periodically, a server constructs an invalidation report $\langle t_0, U\_list, ctnc\rangle$, where t_0 is a timestamp less than *ctnc* and *U_list* is the identities of the set of data items which has been updated between t_0 and *ctnc*, i.e., $U\_list = \{x | \exists x^k \; s.t. \; t_0 < k \leq ctnc\}$. The invalidation report is broadcasted to every mobile computer in the cell. When a mobile computer receives the validation report, it can advance the timestamp of its cache to *ctnc* after dropping all the data items in *U_list*, if the original timestamp of the cache is between t_0 and *ctnc*. The correctness of this approach is shown in the following lemma.

Lemma 6 *Suppose V is a consistent version set with timestamp t, where $t_0 \leq t \leq ctnc$. Let V' be the version set obtained by removing all the data items in U_list from V. Then V' is a consistent version set with timestamp $ctnc$.*

Proof: Let $U\_version = \{x_i^j | x_i^j \in V \wedge x_i \in U\_list\}$ (the data items in the cache that have been updated in the system between t_0 and $ctnc$) and $W = \{x_i^j | x_i^j \in V \wedge \exists x_i^k \ s.t. \ t < k \leq ctnc\}$ (the data item in the cache that have been updated in the system between t and $ctnc$). Obviously, $W \subseteq U\_version$, let $T = U\_version - W$. By Lemma 5, $V - W$ is a consistent version set with timestamp $ctnc$. By Lemma 3, $V' = V - W - T$, is also a consistent version set with timestamp $ctnc$. Hence the lemma follows. ∎

If the timestamp of a cache is smaller than t_0, Lemma 6 cannot apply. Obviously, from Definition 2, an empty version set is a consistent version set with any timestamp. Therefore, when the timestamp of a cache is smaller than t_0, the mobile computer can drop all the data items in its cache and advance the timestamp to $ctnc$. Hence, the range of t_0 and $ctnc$ must be carefully chosen so that not too many mobile computers are required to drop all the data items in their cache. However, sometimes it is very difficult to choose a suitable timestamp t_0, especially when the timestamps of the mobile computers in a cell is too diversified. When the range of t_0 and $ctnc$ is too small, most of the mobile computers will drop all the data items from their cache. When the range is too large, U_list will contain data items from a large portion of the database. As a result, a significant amount of data items in a cache has to be dropped. In order to alleviate this situation, the format of an invalidation report can be modified as follows:

$\langle t_0, U_0, t_1, U_1, \cdots t_j, U_j, t_{j+1} = ctnc \rangle$, where

1. $t_0, t_1 \cdots t_k$ are timestamps such that $t_0 < t_1 < \cdots t_k < t_{j+1} = ctnc$,
2. $U_j = \{x | \exists x^k \ s.t. \ t_j < k \leq ctnc\}$, and
3. for $0 < l \leq j-1, U_l = \{x | \exists x^k \ s.t. \ t_l < k \leq ctnc\} - \cup_{l<m\leq j} U_m$.

The intuitive idea of the design is that $U_l \cup U_{l+1} \cup \cdots \cup U_j$ contains all the data items that have been updated between timestamps t_l and $ctnc$. Therefore, a mobile computer can advance its timestamp to $ctnc$ by dropping all these data items, if its original timestamp is between t_l and $t_l + 1$. The algorithm to invalidate data items is depicted in Figure 3, where t is the timestamp of the cache. Note that the algorithm is required to be executed atomically with respect to the read-only transactions in the mobile computer.

```
Procedure ReceiveInvalidate
t: the timestamp of the cache
begin
    Suppose an invalidation report ⟨t_0, U_0, t_1, U_1, ··· t_j, U_j, t_{j+1} = ctnc⟩ is received
    If t_0 > t
        drop the entire cache
        t ← ctnc
    else if ctnc < t
        ignore the invalidation report
    else
        For l from 0 to j
            if t_l ≤ t < t_{l+1}
                For each data x ∈ ∪_{l<i≤ctnc} U_i
                    Drop x from the cache
                t ← ctnc
                Return
end
```

Figure 3: Algorithm for Processing Invalidation Report at a Client

4.4 Data Broadcasting

After an invalidation report is broadcasted, most of the mobile computers in the cell will drop some of its data items. These mobile computers may then request new versions of these data items at almost the same time. Such a sudden increase in the number of requests may jam the system. In order to alleviate this situation, the server can broadcast some frequently requested data items immediately after an invalidation report has been broadcasted. The format of a $DATA$ message is as follows:

$$\left\langle ctnc, x_{i_1}^{j_1}, val_{i_1}, \cdots x_{i_n}^{j_n}, val_{i_m} \right\rangle,$$

where x_i^j are the latest version of the data item x_i with version number smaller than $ctnc$ and val_i is the value associated with the version. Note that it is not difficult to merge a $DATA$ message with an invalidation report so that some redundancy can be removed. However, for the sake of explanation, we discuss invalidation reports and $DATA$ message separately. Upon receiving a $DATA$ message from the local server, a mobile computer executes the algorithm in

Fig 4 for loading new data items into its cache.

Procedure ReceiveDATA

begin

  Suppose a $DATA$ message $\left\langle ctnc, x_{i_1}^{j_1}, val_{i_1}, \cdots x_{i_n}^{j_n}, val_{i_m} \right\rangle$ is received.

  If $ctnc < t$

    ignore the $DATA$ message

  else

    For each data version x_i^j in the $DATA$ message

      If $t \geq j$ and the mobile computer needs the data item

        Cache x_i^j

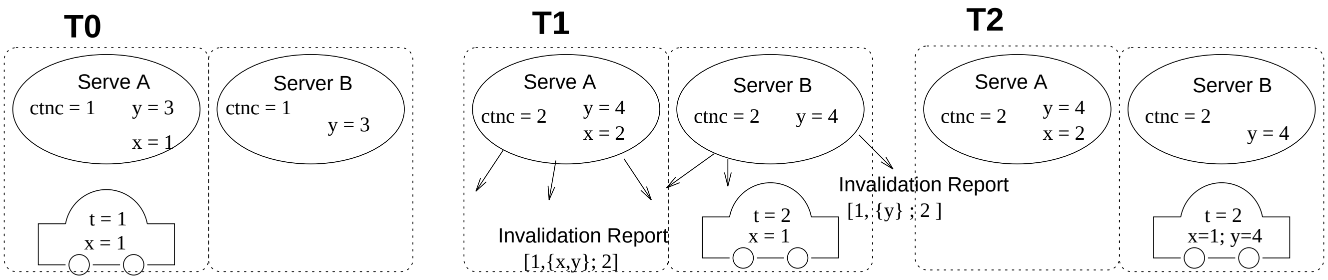
end

Figure 4: Algorithm for processing Data Broadcasted from a Server

Note that although a $DATA$ message is broadcasted immediately after an invalidation report, we cannot assume that the timestamp, t, of the mobile computer is $ctnc$. This is because the mobile computer may fail to receive the invalidation report or the mobile computer ignored the invalidation report because the timestamp of the mobile computer is greater than the ctnc of the server. In the algorithm, if $ctnc < t$, the $DATA$ message will be ignored. Otherwise, the mobile computer can load a data item if t is greater than the version of the data item. It is because for each data version x_i^j in the $DATA$ message, x_i^j must be the latest version with version number less than or equal to $ctnc$. Hence, if $ctnc \geq t \geq j$, x_i^j is also the latest version with version number less than or equal to t. Therefore, by Lemma 2, the mobile computer can load the data item into its cache without destroying cache consistency.

5 Partially Replicated Database System

So far we have assumed that databases are fully replicated, i.e., every server stores a copy of each data item. In practice however, some location sensitive data item may not be fully replicated, since these data items may not be useful for mobile computers in every cell. In such case, the database is referred to as a partially replicated database. The issue of maintaining cache consistency for a partially replicated database is a little bit more complicated, since a server only manages information for its local copies. Therefore, the invalidation reports of a server will never contain data items that are not supported by the server. Problems may arise when a mobile computer carries some data items to the cell of a server that does not support the data items. Since the data items will never be invalidated, cache consistency may be destroyed when the mobile computer advances its timestamp. Such a scenario is illustrated in Figure 5.



The values of data at different sequence version:

version number	x	y
1	1	3
2	2	4

At Time T_0 — The mobile computer loads x from server A and sets its timestamp, t, to 1.

At Time T_1 — The mobile computer crosses the boundary to server B, at the same time both x and y are updated by a transaction. These events are indicated in the invalidation reports which are sent afterwards. However, the mobile computer at server B is not informed about the update of data item x, but the timestamp of the cache advances to 2.

At Time T_2 — Cache consistency is destroyed when the mobile computer loads y from server B.

Figure 5: A scenario for a Partially Replicated Database

In order to solve this problem, we suggest to attach a *server-list* to each non-fully replicated data item, where the list contains the identity numbers of the servers which support the data item. Whenever a non-fully replicated data item is loaded in a cache, the server-list will also be stored in the cache. In addition, each server is required to attach its identity number to every invalidation report. When a mobile computer receives an invalidation report, it can compare the identity of the server with those in the server-list of each non-fully replicated data item. A data item has to be dropped, if it is not supported by the local server. After non-fully replicated data items have been considered, the normal algorithm in Figure 3 can be used to invalidate fully replicated data items and advance the timestamp.

When a mobile computer requests a data item that is not supported by the local server, the local server can forward the request to a server which contains the data item and possesses the latest version with version number less than the timestamp of the mobile computer (based on $V[].ctnc$). Then the response which contains the value and server-list of the data item will be forward to the mobile computer through the local server. However, by using the above invalidation mechanism, it is expected that the data item will be dropped, once a new invalidation report is received.

When the frequency for accessing a data item that is not supported in a local server is high, the server can decide to support the data item. The server is only required to add its identity number to the server-list of the data item and then propagate this modification to other servers. No special synchronization is needed. A mobile computer which carries the data item with an outdated server-list will drop the data item when it first receives an invalidation report from the server. However, it can load the data item (with the updated server-list) again by requesting it from the server or when it recieves a $DATA$ message which contains the data item.

Similarly, when the frequency for accessing a data item that is supported in a local server is low, the server can decide not to support the data item. Firstly, the server needs to modify the server-list of the data item. Secondly, a new version of the data item is created as if a write operation had been executed. Similar to an update transaction, this "write" operation also updates the server-list of the copies in the write quorum. Therefore, simultaneous updates of the server-list is avoided by the replica control protocol. This new version together with the new server-list will be then be propagated to other servers not in the write quorum by the version control mechanism. By using this approach, the successive invalidation reports will contain the identity of this data item, until the version number is less than t_0 of the invalidation reports. Therefore, a mobile computer which carries the data item with an outdated server-list will drop the data item, once it receives an invalidation report from the server.

6 Conclusion and Future Work

Performance is the main concern for most of the previous work on the caching issues in a mobile computing environment. These results are mainly based on the property that communication in a mobile computing environment is expensive and unreliable. Relatively, the impact of the unrestricted mobility of mobile computers on data consistency is seldom addressed. In an attempt to complement the previous results, the notion of cache consistency for mobile computers is defined and the issues for maintaining cache consistency are investigated. In addition, a new caching policy for mobile computers is proposed. The new caching policy not only reduces the contention on the narrow bandwidth wireless network, but also maintains cache consistency. By using this new policy, the data in the cache of a mobile computer must correspond to a snapshot of a database. Read-only transactions can safely read data from the cache without issuing any lock to the servers and at the same time serializability is guaranteed.

The policy introduced in this paper is by no means perfect. The present work suggests some continuations. Some applications may not require to adopt serializability as the correctness criterion. If the correctness criterion can be relaxed, a weaker notion of cache consistency can be used. As a result, the synchronization constraints for cache management may be relaxed. Another issue that may be worth investigating is to reduce the length of invalidation reports. In the current approach, an access unit is also an invalidation unit. If the database is large and updates are frequent, an invalidation report may consume a large amount of bandwidth. The length of invalidation reports may be reduced, if we can group several data items as one invalidation unit. However, the way to group the data items and the impact on the performance need further investigation. Currently, we are studying the performance and behavior of the proposed caching policy by using simulation. We hope that the result will benefit the design of the cache management policy in the future mobile computers.

References

[1] A. Acharya and B. R. Badrinath. Delivering Multicast Messages in Networks with Mobile Hosts. In *Proceedings the 13th International Conference on Distributed Computing Systems*, pages 292–299, 1993.

[2] B.R. Badrinath and T. Imielinski. Replication and Mobility. In *Second Workshop on the Management of Replicated Data*, pages 9–12, 1992.

[3] D. Barbara and T. Imielinski. Sleepers and Workaholics: Caching Strategies in Mobile Environment. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 1–24, 1994.

[4] P. A. Bernstein and N. Goodman. Multiversion Concurrency Control – Theory and Algorithms. In *ACM Transactions on Database Systems, Vol 8, No. 4*, pages 465–483, 1983.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, Reading, Massachusetts, 1987.

[6] S.K. Feiner D. Duchamp and G.Q. Maguire. Software Technology for Wireless Mobile Computing. In *IEEE Network Vol. 5. No.6*, pages 12–18, 1991.

[7] D. Duchamp. Issues in Wireless Mobile Computing. In *Proceedings. Third Workshop on Workstation Operating Systems*, pages 2–10, 1992.

[8] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, November 1976.

[9] G.H. Forman and J. Zahorjan. The Challenges of Mobile Computing . In *Computer Vol.27 No.4*, pages 38–47, 1994.

[10] D. K. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, pages 150–159, December 1979.

[11] Y. Huang, P. Sistla, and O. Wolfson. Data Replication for Mobile Computers. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 13–24, 1994.

[12] T. Imielinski and B. R. Badrinath. Querying in highly mobile distributed environments. In *Proceedings of the 18th VLDB Conference Vancouver, British Columbia, Canada*, pages 41–52, 1992.

[13] E. Levy, H. F. Korth, and A. Silberschatz. An Optimistic Commit Protocol for Distributed Transaction Management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 88–97, May 1991.

[14] P.A. Bernstein, V. Hadzilacos and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley, 1987.

[15] D. P. Reed. Naming and Synchronization in a Decentralized Computer System. Technical Report MIT-LCS-TR-205, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1978.

[16] M. Satyanarayanan. Mobile Computing. In *Computer Vol. 26. No.9*, pages 81–82, 1993.

[17] O.T. Satyanarayanan and D. Agrawal. Efficient Execution of Read-Only Transactions in Replicated Multiversion Databases. In *IEEE Transactions on Knowledge and Data Engineering, Vol 5, No 5*, pages 849–871, 1993.

[18] D. Skeen. Non-blocking commit protocols. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 133–147, June 1982.

[19] G. T. Wuu and A. J. Bernstein. Efficient Solutions to the Replicated Log and Dictionary Problems. In *Proceedings of the Third ACM Symposium on Principles of Distributed Computing*, pages 233–242, August 1984.