Autonomous Transaction Processing Using Data Dependency in Mobile Environments
by I Chung, B Bhargava, M Mahoui, L Lilien
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

Autonomous Transaction Processing Using Data Dependency in Mobile Environments[*]

IlYoung Chung,[1][†] Bharat Bhargava,[1] Malika Mahoui,[2] and Leszek Lilien[1]

[1] Department of Computer Sciences
and Center for Education and Research in Information Assurance and Security (CERIAS)
Purdue University
West Lafayette, IN 47907
iychung@yahoo.com, {bb, llilien}@cs.purdue.edu

[2] Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

mmahoui@cis.upenn.edu

Abstract

Mobile clients retrieve and update databases at servers. They use transactions in order to ensure the consistency of shared data in the presence of concurrent accesses. Transaction processing at mobile clients faces new challenges to accommodate the limitations of mobile environments, such as frequent disconnections and low bandwidth. Caching of frequently accessed data in a mobile computer can be an effective approach to continue transactions in the presence of disconnections or other reasons for losing messages. It can help to reduce contentions on the narrow bandwidths of wireless channels. Concurrency control schemes using caching ensure consistency among data items from the server and from the client caches. We present a scheme that can increase the autonomy of mobile clients for validating transactions, using caching and pull-based data delivery. In the scheme, mobile clients can decide to commit read-only transactions locally, without interaction with the server and can detect transaction aborts earlier. The clients receive from the server dependency information, from which they build partial serialization graphs. Dependency information is based on the notion of i-order dependency introduced in the paper. We study the performance of the proposed protocol by means of simulation experiments.

1   Introduction

The population of mobile users (clients) as well as the number of multidatabase systems (servers) offering services that allow mobile clients to access and to share data continues to grow [12]. Transaction processing has faced new challenges to accommodate the limitations of a mobile computing environment. Mobile computing is characterized by low bandwidth and message loss due to disconnections and lost, misrouted or erroneous packets. Mobile hosts should be more autonomous in the management of mobile transactions [12]. Availability of more powerful mobile devices with moderate storage capacity allows for caching of frequently accessed data on a mobile client and managing data locally [6, 7]. Caching provides mobile computers with more local autonomy to deal with message losses and to reduce bandwidth contention. However, a caching strategy needs to be combined with an invalidation strategy to ensure that locally stored data are consistent with data stored at the server. Another factor, which needs to be considered when devising a transaction processing technique, is minimizing the period during which a transaction execution is blocked due to a validation of its execution at the server. This is not only because of limited resources, such as power, but also due to the real-time nature of many mobile transactions. Therefore, a transaction that cannot be committed should be detected and restarted as early as possible . Transactions that can be committed locally at the mobile client without involving the server should be identified.

We propose a new transaction processing protocol that aims to provide more local autonomy to mobile clients. In the proposed scheme, mobile computers can locally commit read-only transactions, and can earlier abort transactions that can not be committed. The protocol combines two approaches for checking serializability of concurrent executions: the optimistic approach applied when a transaction accesses a cached copy of a data item, and the pessimistic approach applied when a transaction accesses a data item requested from the server. The pessimistic approach relies on the partial view of the serialization graph provided by the server with the requested data. The partial view is used to commit read-only transactions locally, and for earlier detection of some non-serializable situations involving update transactions. Update transactions that reach the end of their execution are sent to the server to be certified.

The proposed protocol does not require each mobile client to build the serialization graph. This would generate overhead in bandwidth utilization and mobile client resources. Rather, mobile clients maintain information about dependencies only with each cached data item. This information is not delivered by the server as a separate message, but is incrementally updated at a mobile client each time a data item is requested from the server. The server piggybacks the dependency information on the reply message containing the requested data item.

Some local commit strategies for a read-only transaction have been proposed in the push-based data dissemination environments, in which data items are broadcast to mobile clients without any explicit request [8, 11, 14]. When the entire data item sets are acquired via the broadcast channel, the local commit strategy can be achieved more easily, since data items delivered in a single cycle are guaranteed to be in a consistent state. However, in the push-based data delivery it is difficult to predict accurately the needs of mobile clients, which results in sending irrelevant data. This results in a poor use of channel bandwidth and data may not reach mobile clients in time. We present an autonomous transaction processing scheme using the pull-based request-response data delivery with caching.

Additional goals for the proposed protocol include relieving the server from the burden of committing all transactions and reducing the number of transaction aborts. The ways to achieve both these goals are discussed in more detail in Section 2. In Section 3 we present the correctness criteria for the protocol and the relevant terminology. In Section 4 we describe and discuss the proposed protocol. Section 5 presents experiments and results. Related research is discussed in Section 6. We conclude the paper in section 7, including some directions for future work.

2 System Model

In a mobile computing environments the network is made up of stationary and mobile hosts. Mobile hosts change location and network connections while computations are being processed. While in motion, mobile hosts retain their network connections through the support of specialized stationary hosts with wireless communications abilities, which are called mobility support stations (MSS). Each MSS is responsible for all mobile hosts within a given area, known as a cell. At any given instance, a mobile host may directly communicate only with the MSS responsible for the area in which it moves.

2.1 Mobile Client Caching

Within the mobile computing environment, shared data are stored and controlled by a number of database servers executing on stationary hosts. Mobile clients have a limited storage capacity for cached data [12]. Caching frequently accessed data at a mobile client is an effective approach to reduce contention on the narrow bandwidths of wireless channels. Mobile clients can cache and store shared data on a hard disk or in a memory which survives power failures or disconnections. Caching allows database systems to use the resources of mobile clients in order to reduce the number of data requests that must be sent to the server [6, 7]. Mobile client caching is a compromise between the utilization of local resources and the correctness and availability concerns. By maintaining cache, the

need to obtain data from the servers is reduced. Data copies that are in a cache can be accessed faster than data stored on the server [13].

Caching does not transfer ownership of data to mobile clients. Servers remain the owners of data. Servers are ultimately responsible for ensuring the correctness of transaction executions. Caching is a compromise in the sense that servers maintain data ownership, and remain involved in transaction execution. As a result, some client autonomy is sacrificed. Minimizing the server's participation in transaction execution is the key to making this compromise to pay off. Reducing the dependency of mobile clients on the server is an objective for our protocol.

2.2   Mobile Transaction Processing

In order to ensure the consistency of shared data in the presence of concurrency and failures, users at mobile clients retrieve and update data using transactions. Each MSS has a coordinator which retrieves transaction operations from mobile clients and monitors their execution in database servers within the fixed networks. Transaction operations are submitted by a mobile client to the coordinator in its MSS, which in turn sends them to the distributed database servers within the fixed network for execution.

In general, there are two ways to structure a mobile transaction processing system. First, the mobile client can behave like a remote I/O device and all data must be placed in the stationary network [7, 12]. This situation arises when the resources of a mobile device are very limited. Second, a mobile host can behave as server, fully managing transactions and storing data locally. We adopt an approach between these two extremes. We store data locally, but we treat them as a cache rather than as a primary copy. Each mobile client has its own cache to maintain some data for later reuse. A transaction operation that accesses a data item stored in the cache can be processed without interaction with the database server. If the data item does not exist in the cache of the mobile host, it requests it form the server.

The server checks whether a transaction initiated at a mobile client satisfies the correctness criteria. A mobile client sends a message requesting this certification for a transaction to the server. Transaction processing techniques can be classified based on the unit of certification as either operation-based or transaction-based. The difference between operation-based and transaction-based approaches lies in the expectations of a mobile client for the future of the initiated transaction. If a mobile client expects conflicts related to the initiated transaction, it adopts a pessimistic approach. This prevents conflicts by a priori checking each operation of the transaction [9]. If a mobile client expects few conflicts, since a transaction will use a few shared data items, it can use the optimistic approach. It

4

means that it executes the entire transaction locally and it sends the entire transaction's history to the server for validation only at the end of the transaction [2, 8, 11, 14].

Since in the optimistic approach a mobile client executes a transaction autonomously until all operations are completed, this approach does not suffer from the communication overhead in the mobile environments. As a result, most of the existing transaction processing techniques for such environments adopted transaction-based certification to reduce the dependency upon the server while executing transactions. Some portions of transaction executions can be transferred to mobile clients to reduce the burden placed by the certification process on the server and on the limited upstream communication bandwidth.

2.3   Asynchronous Broadcasting

In the transaction-based certification approach, the server broadcasts the list of updated data items along with the results of the certification process. Most of earlier transaction-based certification schemes adopted periodic broadcast for the messages [2, 13]. The server broadcasts the list of data items that should be invalidated in the caches and the list of transactions that were committed in the last period.

This approach, called synchronous broadcasting, has the following weaknesses due to its periodic nature [5]:

- If multiple conflicting transactions have requested commit during the same period, only one of them can commit, and the others have to abort. When more than two transactions have accessed common data items and have sent messages requesting to be committed during the same period, the server can commit only one of them. It results in a high abort rate.

- A mobile client is blocked on the average for the half of the broadcast period, until it can either commit or abort a transaction. A transaction which requested to be committed immediately after a broadcast point can be blocked for a long period until the next broadcast. This can degrade the throughput.

- When a large portion of data items are updated in the same interval, the rate of aborts will increase due to the larger rate of inconsistent cached copies. Because updates of data items are sent to mobile clients only at the next broadcast point, the cached copies remain stale until then. As a result, transactions that accessed these copies will be aborted.

- When there are few updates, the periodic delivery of consistency information can be an unnecessary communication overhead. The server sends broadcast messages, which include the list

of updated data items, independent of the frequency of updates. If there are no updates during a broadcast period, an empty set of updates will be broadcast. When mobile clients can certify read-only transactions autonomously, the periodic broadcast is unnecessary.

We adopt the asynchronous broadcast approach as the way of sending control messages. Unlike schemes that use synchronous broadcast, the control messages are broadcast by the server immediately after a commit decision is made by the server. The asynchronous approach can avoid many aborts which occur in the synchronous approach due to the conflicts within the same broadcast period. Aborts due to a late notification of mobile clients about updates can be reduced. The waiting time for a transaction that requested the server for commitment can be reduced, since the server immediately broadcasts the commit-or-abort result of the certification to the transaction [5].

3   Data Dependency Information

We present here the correctness criterion for the proposed transaction processing protocol. We assume that there is a central server that holds and manages all the data. Updates can be generated not only by the server but also by mobile clients. They have to be ultimately installed in the server's database when committed. Mobile clients request data items from the server when client applications request them. Mobile clients maintain cached copies of data to speed up the reading process.

The proposed protocol uses a modified version of the optimistic (transaction-based) concurrency control in order to reduce the communication costs over the wireless network [2, 8, 11, 14]. In the optimistic approach, all operations of a transaction can be processed locally at mobile clients, either using cached data items or requesting them from the server. The transaction scheduler checks at every transaction's commit point whether the execution history that includes the transaction is serializable or not. Unless it is serializable, the transaction is aborted. There are many ways to implement an optimistic concurrency control protocol, and in this paper we focus on the mechanism that uses the serializability graph [3, 4].

Each data item in the system is tagged with a timestamp that uniquely identifies the state of the data. The timestamp associated with a data item is increased by the server when a transaction which updates the data item is committed. All data items updated by a transaction are given the same timestamp. If a data item x is updated by T_i, the timestamp of x is referred as $ts(x)$, and has the same value as $ts(T_i)$. Conflicts between transactions can be defined using the following definition.

Definition 1 *Let T_i and T_j be two transactions. We say that T_i (directly) conflicts with T_j or that T_i (directly) precedes T_j, iff there exists a data item x such that:*
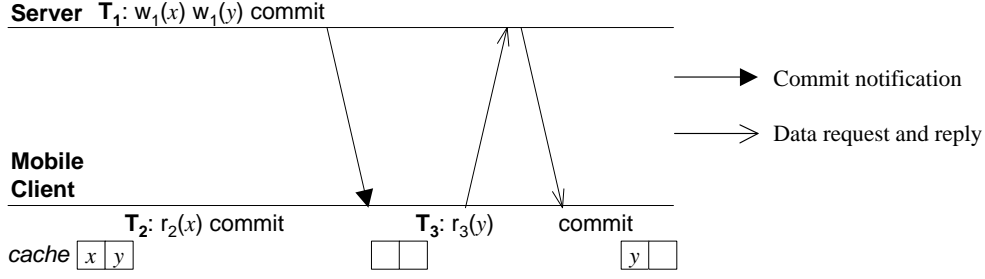
Figure 1: Conflict relation between transactions - an example.

1. T_i *has performed a read/write operation on data item x with ts(x) = t1, and T_j performs a write operation on x with ts(x) = t2, and t1 < t2, or*

2. T_i *has performed a write operation on data item x with ts(x) = t1, and T_j performs a read operation on x with ts(x) = t2, and t1 $\leq$ t2.*

If T_i directly precedes T_j, then T_i precedes T_j in the serialization order. This is represented by an edge $T_i \to T_j$ in the serialization graph. Let's see a simple example execution of transactions in Figure 1. In this paper, $r_i(x)$ and $w_i(x)$ denote a read and a write operation, respectively, performed by transaction T_i on data item x.
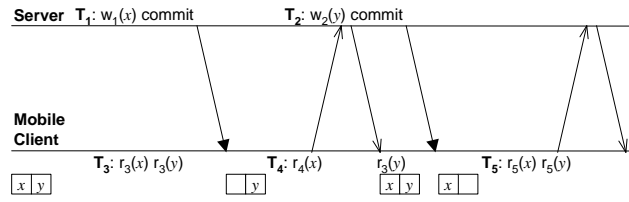
Since transaction T_2 reads a copy of x that was stored in the cache, its timestamp is lower than that of the data item x in the server, which was updated by transaction T_1. As a result, T_2 precedes T_1 in the serialization order. On the other hand, T_3 follows T_1 because T_3 has read data item y with the timestamp equal to that of the server's original copy of y, which means that T_3 read the value of y written by T_1. As a result, $T_2 \to T_1 \to T_3$.

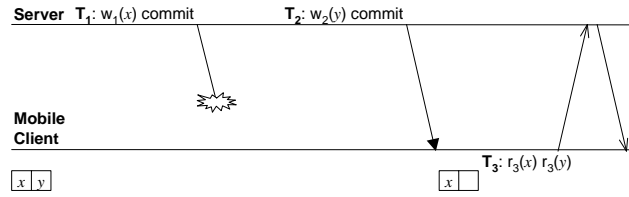There can also exist indirect conflict relations between transactions, which can be defined as follows.

Definition 2 T_i *w-precedes T_j if there exist $w - 1$ transactions T_1, T_2,..., T_{w-1}, such that $T_i \to T_1 \to T_2 \to ... \to T_{w-1} \to T_j$.*

By default, T_i 1-precedes T_j simply means that T_i directly precedes T_j, which corresponds to the direct serialization order.

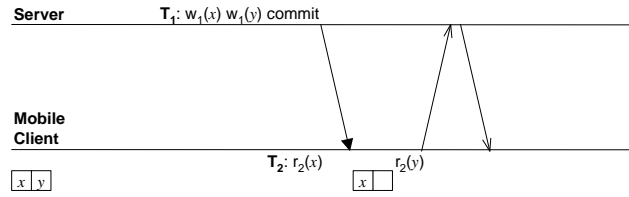The serialization graph can be produced with transactions initiated by multiple mobile clients, and we can attain serializable execution by ensuring that this graph always remains acyclic [3, 4]. This is the main tool adopted in this paper to ensure the correctness of the concurrency control algorithms, which allow local commit of read-only transactions that access cached copies at mobile clients.
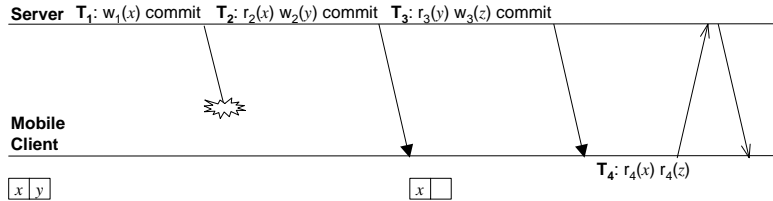
Server   **T₁**: w₁(x) commit   **T₂**: w₂(y) commit

Mobile
Client
        T₃: r₃(x) r₃(y)   **T₄**: r₄(x)   r₃(y)   **T₅**: r₅(x) r₅(y)

$x \mid y$ $\boxed{\ \mid y}$ $x \mid y$ $x \mid \ $

(a) Example 1

Server   **T₁**: w₁(x) commit   **T₂**: w₂(y) commit

Mobile
Client
                                                          T₃: r₃(x) r₃(y)

$x \mid y$ $x \mid \ $

(b) Example 2

Server   **T₁**: w₁(x) w₁(y) commit

Mobile
Client
              T₂: r₂(x)   r₂(y)

$x \mid y$ $x \mid \ $

(c) Example 3

Server **T₁**: w₁(x) commit   **T₂**: r₂(x) w₂(y) commit   **T₃**: r₃(y) w₃(z) commit

Mobile
Client
                                                         T₄: r₄(x) r₄(z)

$x \mid y$ $x \mid \ $

(d) Example 4

Figure 2: Example scenarios of transaction executions.

To illustrate this, let us consider the example transaction execution scenarios shown in Figure 2. We assume that at the beginning of each example mobile clients have copies of data items x and y in their caches. In example (a), T_3 precedes T_1 and T_2, since it reads data items x and y whose timestamps are lower than those written by T_1 and T_2, respectively. T_1 is followed by T_4, since T_4 reads the value of data item x written by T_1. In this case, the timestamp of data item x written at the server by T_1, and the timestamp of x read by T_4 are equal. T_5 follows both T_1 and T_2, since it reads data items whose values have been updated by T_1 and T_2. As a result, we can get the serialization graph $T_3 \rightarrow T_1 \rightarrow T_4 \rightarrow T_2 \rightarrow T_5$.

In example (b), the mobile client never received the server's broadcast message which notified

8

about an update of x. This message loss could be due to a number of reasons, including client disconnection, lost packets, errors in packets, or misrouted packets. T_3 reads an outdated copy of x from the cache. As a result, T_3 precedes T_1 in the serialization order. W.r.t. data item y, T_3 follows T_2, since it read data item y whose timestamp is equal to that of T_2. Thus, the serialization graph becomes $T_3 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3$, which includes a cycle.

In example (c), the mobile client receives the broadcast message that invalidates data items x and y during execution of transaction T_2. Since T_2 has already read from the cache x with the timestamp $ts(x)$ lower than the one written by T_1, $T_2 \rightarrow T_1$. However, before T_2 reads y, its cached value is invalidated by the commit notification message for T_1. The client must request for the new value of y. This produces the conflict relationship $T_1 \rightarrow T_2$, which creates a cycle in the serialization graph.

Example (d) shows a case in which a cycle is produced on the server by indirect conflicts between transactions. The mobile client never received the notification about the update of x by T_1. Therefore, T_4 reads the outdated cached copy of x. Hence $T_4 \rightarrow T_1$. However, T_4 reads the most recent value of z that was updated by T_3. Hence $T_3 \rightarrow T_4$. Although there is no direct conflict relation between T_1 and T_3, committing T_4 creates a cycle in the serialization graph, that is $T_4 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$.

We present a scheme that delivers needed serialization graph information to each mobile client. With this information, mobile clients can decide whether committing a locally completed read-only transaction would or wouldn't produce a cycle in the serialization graph. Mobile clients can decide either to commit or abort a read-only transaction autonomously, without submitting it to the server. For update transactions, a mobile client cannot detect all cycles caused by such transactions without submitting them to the server. However, using conflict information received from the server, it can detect early the necessity to abort most conflicting update transactions. As a result, upstream bandwidth and energy consumption in mobile clients are saved. Dependency information is not delivered by the server as a separate message, but is piggybacked on data delivery messages and is incrementally updated at mobile clients each time a data object is received from the server.

In order to describe serialization graph information delivered to mobile clients, we define the notion of order dependency between data items.

Definition 3 *Let x and y be two data items. We define the w-order dependency relation $\ll_{w,t}$ between data items x and y as follows:*

- *$w = 0$: $x \ll_{0,t} y$, iff there exists a transaction T_i which performs write operations on both x and y such that $ts(T_i)=t$*

- *$w \geq 1$: $x \ll_{w,t} y$, iff there exist at least two transactions T_i and T_j, such that*

- T_i *has performed a write operation on data item* x,

- T_j *has performed a write operation on data item* y,

- T_i *w-precedes and conflicts with* T_j, *and*

- *ts(*T_j*)=t*

Lemma 1 *Suppose that read-only transaction* T_k *at a mobile client accesses data item* y, *which was updated by a committed transaction* T_j *with the timestamp ts(*T_j*)= t.* T_k *is involved in a cycle with* T_j *only if* T_k *accesses any data item* x *such that* $x \ll_{w,t} y$ *and* $ts(x) < t$.

Proof. Since T_k reads y which was updated by T_j, there exists a conflict relation between T_j and T_k, that is $T_j \rightarrow T_k$. To have T_k involved in a cycle with T_j, there should be a transaction T_i that show a conflict relation $T_k \rightarrow T_i \rightarrow ... \rightarrow T_j$ (T_i can be T_j if there exists a cycle between the two transactions). Since T_k is a read-only transaction, to have a conflict relation $T_k \rightarrow T_i$, T_k must read data item x whose timestamp is lower than ts(T_i). On the other hand, there exists the relation $T_i \rightarrow ... \rightarrow T_j$, T_i w-precedes T_j, and as a result $x \ll_{w,t} y$. $\square$

Let us revisit the example execution scenarios of Figure 2. In example (c), if ts(T_1) = 1, $x \ll_{0,1} y$ (or $y \ll_{0,1} x$) by the definition. Since T_2 accesses two data items that show dependency relation, by Lemma 1 we should check whether committing T_2 produces a cycle in the serialization graph. In the example, T_2 should be aborted, since it read x with the timestamp which is lower than $ts(T_1)$, and y with the timestamp larger than $ts(T_1)$. This produces a cycle. In example (d), $x \ll_{2,3} z$, if $ts(T_3)$ = 3. Since T_4 reads x and z that are dependent, we have to check if a cycle is produced when T_4 is committed. In the example, due to a lost commitment message T_4 reads x with the timestamp lower than the timestamp which can be seen by T_3. As a result, it must not be committed.

To define conflict information to be downloaded to mobile clients, we specify the list of dependents for a data item.

Definition 4 *Let* y *be a data item and let* t *be the timestamp of a transaction that updates* y. *We define the list of dependents of* y *with timestamp t for window size* w, *as follows:*

$Dependent_{w,t}(y) = \{x \mid (x \ll_{i,t} y),$ *where i is the lowest value within [0,w] * \}*
(w is the window size for the i-order dependencies used to define $Dependent_{w,t}(y))$

The dependency list $Dependent_{w,t}(y)$ includes all data items that were updated by transactions that at most w-preceded the transaction which updated y. In other words, $Dependent_{w,t}(y)$ ignores all i-order dependencies for $i > w$.

In our protocol, to detect any cycle related to a read-only transaction being executed in a mobile client, the dependency list $Dependent_{w,t}(y)$ is piggybacked on the data item y, transferred in response to the request from the mobile client. $Dependent_{w,t}(y)$ includes every data item x which satisfies $x \ll_{w,t} y$ complete with its timestamp $ts(x)$. Hence, by Lemma 1, a mobile client can detect if any cycle is produced by a read-only transaction.

4   The Proposed Protocol

The proposed protocol combines two approaches for checking serializability: the optimistic approach, applied when a transaction accesses a cached copy of a data item, and the pessimistic approach, applied when a transaction accesses a data item requested from the server. When a needed data item is available in the cache, the transaction can read it. No validation process is performed. In order to execute an operation on a data item x that is not available at the mobile client or was invalidated by a commit notification, the client requests a copy of x from the server. Upon receiving data item x with control information ($Dependent_{w,t}(x)$), the client checks whether the transaction T accessing x does not generate a cycle in G_T^*, the serialization graph composed of G^* and T, in which G^* is the serialization graph for already committed update transactions. With this strategy, the client can detect conflict situations earlier without interacting with the server, and can locally commit read-only transactions. Committing read-only transactions at mobile clients has several advantages, such as reducing transaction commitment delays at the clients, reducing communication costs, and limiting the processing load of the server [8, 10, 12].

Information maintained at the server   Before describing the algorithms used at the mobile client and at the server, we describe information delivered to and managed by the clients and procedures for updating this information.

- $invalidation\_list$: It includes the list of all data items updated by a transaction, and is attached to the $commit\_notification$ message broadcast by the server when the transaction is committed.

- $Dependent_{w,t}(y)$: $\{x \mid (x \ll_{i,t} y),$ where i is the highest value within $[0,w]$ $\}$

Information created/maintained at mobile clients   This information includes the following.

- $cache$: A cache of a mobile client includes data items which have already been used by the client for execution of a local transaction. Each data item x in the cache, $cache.x$, has its

timestamp, $cache.x.ts$, which was delivered with x from the server. We use $server.x.ts$ to denote the timestamp of a data item x in the server.

- $read\_set$: This is the list of data items that are read by a transaction with their timestamps. The timestamp of an item x in $read\_set$ is referred to as $read\_set.x.ts$.

- $write\_set$: This is the list of data items which are written by a transaction with their timestamps. The timestamp of an item x in $write\_set$ is referred to as $write\_set.x.ts$.

- $commit\_request$: A mobile client sends this message to the server when an update transaction T_i is completed. This message includes the id of T_i, its $read\_set$ and $write\_set$.

Algorithm at mobile clients   We now present our algorithm for mobile clients.

MA. Transaction execution state

1. Initially each transaction is marked as *read-only*.

2. When a read-only transaction requests its first write operation, its state is changed to *update* transaction.

MB. Cache update

1. Whenever the mobile client receives a $commit\_notification$ message, it removes the cached copies of data items that are identified by the $invalidation\_list$.

2. When a transaction T_i requests data item x, it receives its value, its timestamp, and the list $Dependent_{w,t}(x)$. For each data item z in the cache, z is removed from the cache if its timestamp $cache.z.ts$ satisfies one of the following conditions:

    - $z \in Dependent_{w,t}(x)$ and $cache.z.ts < Dependent_{w,t}(x).z.ts$, or

    - $z \notin Dependent_{w,t}(x)$ and $cache.z.ts < t\_min$, where
      $t\_min = \{\text{minimum } Dependent_{w,t}(x).y.ts \mid y \in Dependent_{w,t}(x) \}$

MC. Read-only transaction processing

1. If x is in the cache, read it and return.

2. If x is not in the cache, request x from the server.

3. Let A $= read\_set \cap Dependent_{w,t}(x)$ and B $= read\_set - Dependent_{w,t}(x)$.

4. For each $y \in$ A, if $read\_set.y.ts \geq Dependent_{w,t}(x).y.ts$, go to 5; else abort and restart the transaction.

5. If for each $y \in$ B, $read\_set.y.ts \geq t\_min$, read data item; else abort and restart the transaction.

6. When a *read-only* transaction reaches the end of the execution, it can be committed locally.

MD. Update transaction processing

1. If x is in the cache, perform write operation on x and return.

2. If x is not in the cache, request x from the server.

3. Let A = $read\_set \cap Dependent_{w,t}(x)$ and C = $write\_set \cap Dependent_{w,t}(x)$.

4. For each $y \in$ A, if $read\_set.y.ts \geq Dependent_{w,t}(x).y.ts$, go to 5; else abort and restart the transaction.

5. For each $y \in$ C, if $write\_set.y.ts \geq Dependent_{w,t}(x).y.ts$, perform operation on data item; else abort and restart the transaction.

6. When an *update* transaction reaches the end of its execution, send a *commit_request* to the server including the identification of the transaction, its *read_set* and *write_set*.

7. When a mobile client receives *commit_notification* (*abort_notification*) for a transaction, it commits (aborts) the transaction.

When a mobile client begins a transaction T_i, the initial state of T_i is *read-only*. It is changed to *update* only if T_i requests for a write operation. As shown in the mobile client algorithm, upon completion of all T_i operations (when it is ready to commit), the client sends to the server the *commit_notification* message only if the state of T_i is *update*. In case of *read-only* transactions, the mobile client decides to commit autonomously without interaction with the server. The decision to commit an *update* transaction must be made by the server, since the mobile client cannot guarantee that the transaction does not produce a cycle in the serialization graph. To make the commit-or-abort decision for an *update* transaction, the server needs all data items in its *read_set* and its *write_set* together with their timestamp.

Algorithm at the server   The server performs the following algorithm when it receives *commit_request* from a mobile client.
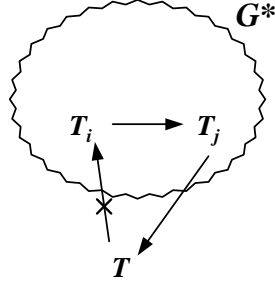
13

Figure 3: Cycle detection in the serialization graph.

1. For each data item $x \in$ *read_set*, if $read\_set.x.ts=server.x.ts$, go to 2; else send *abort_notification* to the mobile client.

2. For each data item $y \in$ *write_set*, if $write\_set.y.ts=server.y.ts$, go to 3; else send *abort_notification* to the mobile client.

3. Insert the identification of the transaction into the *commit_notification* message.

4. Install the values of data items included in the *write_set* in the server database and increment the timestamp.

5. Build the *invalidation_list* by inserting into it each data item from the *write_set* along with its newly computed timestamp, Add this list to the *commit_notification* message.

6. Broadcast the *commit_notification* message.

The server decides to either commit or abort an *update* transaction T_U by comparing timestamps of data items in its *read_set* and its *write_set* with those maintained in the server. If all of them are consistent with the server's, the server can conclude that T_U has executed its operations after receiving *commit_notification* for conflicting transactions, which precede it in the serialization order. Thus, the server can commit T_U. On the other hand, if the timestamp of any data item in the *read_set* or the *write_set* of T_U is lower than the server's, the server must abort the transaction. This is due to the fact that the mobile client executed T_U without knowing the results of one o more conflicting transactions, which precede T_U in the serialization order.

Discussion   The protocol uses dependency information to check whether execution of transaction T does not introduce a cycle in G_T^*, which is the serialization graph composed of G^* and T. Consider the conflicts in Figure 3. If there exist two committed transactions T_i and T_j in G^* such that T_i w-precedes T_j ($T_i \rightarrow_w T_j$), then transaction T may introduce a cycle if there exist conflict relations T
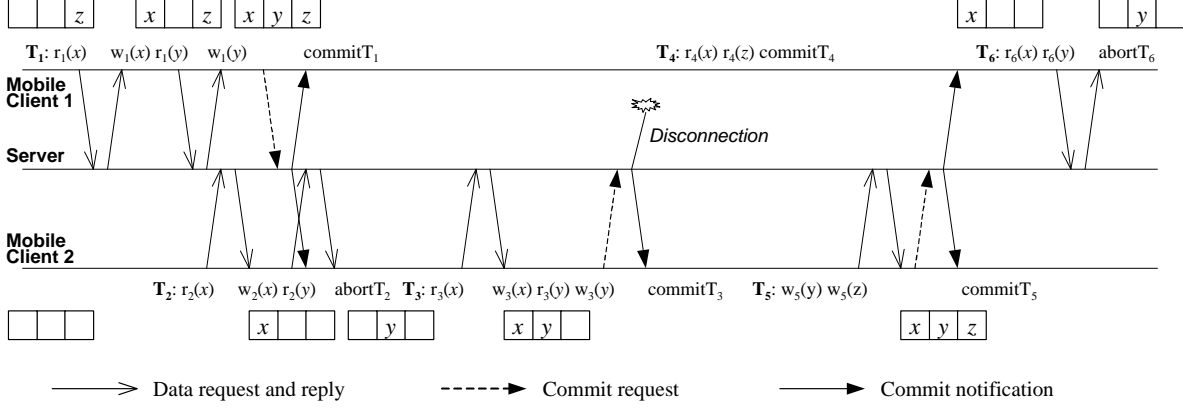
Figure 4: Example execution scenarios for the proposed protocol.

$\rightarrow T_i$ and $T_j \rightarrow T$. The protocol prevents this by using the dependency information. The check for a cycle is performed when the mobile client accesses data item which is not maintained in its cache (see Lemma 1).

Let us assume that T_i and T_j have updated data items x and y, respectively, and T reads both these data items. To create the conflict relation $T \rightarrow T_i$ in G^*_T, T should read a cached copy of x such that $cache.x.ts < \text{ts}(T_i)$. In turn, to create the conflict relation $T_j \rightarrow T$, T should read data item y whose timestamp is greater than or equal to $\text{ts}(T_j)$. This constraint is satisfied when the mobile client requests data item y from the server. If T has already read data item x from the cache, and if $cache.x.ts$ is lower than $\text{ts}(T_i)$, we need to consider two cases. First, if the window size of the $Dependent_{w,t}(y)$ includes the dependency information for transaction T_i, the mobile client can abort T when it finds out that $read\_set.x.ts$ is lower than $Dependent_{w,t}(y).x.ts$. Second, if T_i goes out of $Dependent_{w,t}(y)$ and the set does not include any dependency information for x, T must be aborted. The reason is that in this case $read\_set.x.ts$ is lower than t_min in $Dependent_{w,t}(y)$. On the other hand, if T reads a cached copy of x after it read y from the server, no cycle can be produced, because all copies in the cache with timestamps lower than $\text{ts}(T_j)$ were removed when y was requested.

Example scenarios   An execution scenario for the protocol is shown in Figure 4. This example uses two mobile clients and a server. Once T_1 and T_2 complete all operations, they have to send the *commit_request* messages to the server since they are update transactions. T_2 cannot commit because it run without knowing about the commitment of the conflicting transaction T_1 that executed concurrently. We assume that the timestamp of a transaction is equal to its id, i.e. $ts(T_i) = i$. When T_1 commits, $server.x.ts$ and $server.y.ts$ are changed to 1. Mobile client 2 aborts T_2 when it finds out that $Dependent_{w,1}(y).x.ts$ is larger than $write\_set.x.ts$ (or $read\_set.x.ts$) for T_2. T_2 is restarted as

T_3, which is committed by the server, and the timestamp of each data item updated by it is increased to 3. Mobile client 1 does not receive the server's *commit_notification* message for T_3. Therefore, mobile client 1 still maintains outdated copies of x and y in its cache, both with timestamps equal to 1. The read-only transaction T_4, which accesses these stale cached copies, can be committed locally, since it does not produce any cycle with the transactions that are already committed. Both $read\_set.x.ts$ and $read\_set.y.ts$ are lower than $ts(T_3)$. As a result, $T_4 \rightarrow T_3$. To continue transaction T_6, mobile client 1 requests for y, since y was invalidated by the *commit_notification* for T_5. T_6 is aborted by mobile client 1, when the client finds out that $read\_set.x.ts = 1$ is lower than $Dependent_{w,5}(y).x.ts = 5$. Committing T_6 would produce a cycle $T_6 \rightarrow T_3 \rightarrow T_5 \rightarrow T_6$, since T_6 precedes T_3 w.r.t. x and follows T_5 w.r.t y.

Protocol correctness   Proof of the server algorithm is trivial. The following Lemmas and Theorem prove correctness of the mobile client algorithm.

Lemma 2 *A read operation performed on a newly requested data item does not introduce a cycle in the G^* serialization graph.*

Proof.   Let z be the data item requested by transaction T_c executing on a mobile client. Let T_j be the last transaction that updated data item z. Then, $T_j \rightarrow T_c$. A read operation on data item z generates a cycle in G^* if at least one of the following situations occurs:

1. There is a dependency $T_c \rightarrow T_j$, caused by another data item shared between two transactions T_i and T_j.

2. There exists at least one transaction T_i such that $T_c \rightarrow T_i \rightarrow ... \rightarrow T_j$.

Case 1: Since T_c is in the *read-only* state, the dependency $T_c \rightarrow T_j$ means that T_c has read at least one data item x updated by T_j before T_j committed. In this case T_c will be aborted since:

$$\{x\} \in read\_set \cap Dependent_{w,t}(z) \text{ and } read\_set.x.ts < Dependent_{w,t}(z).x.ts$$

Case 2: The dependency $T_c \rightarrow T_i$ means that T_c read at least one data item x updated by T_i before T_i committed. Therefore

$$read\_set.x.ts < ts(T_i) \tag{1}$$

a. If $\{x\} \in Dependent_{w,t}(z)$, then $\{x\} \in read\_set \cap Dependent_{w,ts}(z)$. Moreover, $T_i \rightarrow_w T_j$ means that

$$ts(T_i) \leq Dependent_{w,t}(z).x.ts \tag{2}$$

16

Using equations (1) and (2), we get

$$read\_set.x.ts < Dependent_{w,t}(z).x.ts \qquad (3)$$

Therefore, T_c will be aborted.

b. If $\{x\} \notin Dependent_{w,t}(z)$, then $w' > w$. Therefore, there exist T_k and w'' such that $T_i \rightarrow_{w''} T_k \rightarrow_w T_j$, and $w' = w + w''$. $T_k \rightarrow_w T_j$ induces

$$ts(T_k) \leq t\_min \qquad (4)$$

Moreover,

$$ts(T_i) \leq ts(T_k) \qquad (5)$$

Using equations (1), (4) and (5), we obtain

$$read\_set.x.ts < Dependent_{w,t}(z).x.ts \qquad (6)$$

Therefore, T_c will be aborted.

$\square$

Lemma 3 *Read operations on data items already present in the cache do not introduce any cycles in the G^* serialization graph.*

Proof. Let T_c' be a transaction composed of all operations of T_c, but rearranged in the order in which they were requested from the server. Since T_c and T_c' are *read-only* transactions, they are equivalent. We will show that if we apply the algorithm (steps 2-5) for each operation of T_c', including read operations from the cache, then the execution of T_c' does not introduce any cycles in G^*. Let *read_set* of T_c' be defined as follows: *read_set* $= \{x_1, x_2, ..., x_i, x_{i+1}, ..., x_k\}$, where x_j $(j \leq i)$ were present in the cache when T_c started its execution.

Case 1: Consider processing of the read operation on x_j, $j \leq i$. Read operation on x_j will be compared with each x_k, for $k < j$. For each x_k we have one of the following conditions satisfied:

$$x_k \in read\_set \cap Dependent_{w,t}(x_j) \wedge read\_set.x_k.ts \geq Dependent_{w,t}(x_j).x_k.ts \qquad (7)$$

or

$$x_k \notin read\_set \cap Dependent_{w,t}(x_j) \wedge read\_set.x_k.ts \geq t_m in \qquad (8)$$

where $t = read\_set.x_j.ts$. If we assume that steps 2-5 of the algorithm are also applied for data items already present in the cache, then for each read operation on data item x_j the read operation will be performed. Therefore, by Lemma 2, the operation does not introduce any cycles.

Case 2: Consider processing of the read operation on x_j ($j > i$). A read operation is checked against each data item x_k ($k < j$). For each such x_k, we have one of the following situations:

- x_k was present in *read_set* when x_j is read. Since x_j was requested by T_c, steps 2-5 of the algorithm were applied for the read operation on x_j. And T_c was not aborted when reading data item x_j.

  or

- x_k was not present in *read_set* when x_j is read This means that x_k was in the cache. After the request of data item x_j, data item x_k satisfied one of the equations (6) or (7). So, if we assume that steps 2-5 of the algorithm are also applied for comparing x_j and x_k, then the operation on x_j will be performed.

Therefore, the execution of x_j does not introduce any cycle in G^*.

$\square$

Theorem 1 *A read-only transaction committed by a mobile client does not introduce any cycles in the serialization graph.*

Proof. No operation of a read-only transaction that was executed by a mobile client produces a cycle in G^* (by Lemma 2 and Lemma 3). $\square$

5 Performance

We develop the simulation model and present the results of experiments to evaluate the performance of the protocol presented in the previous section.
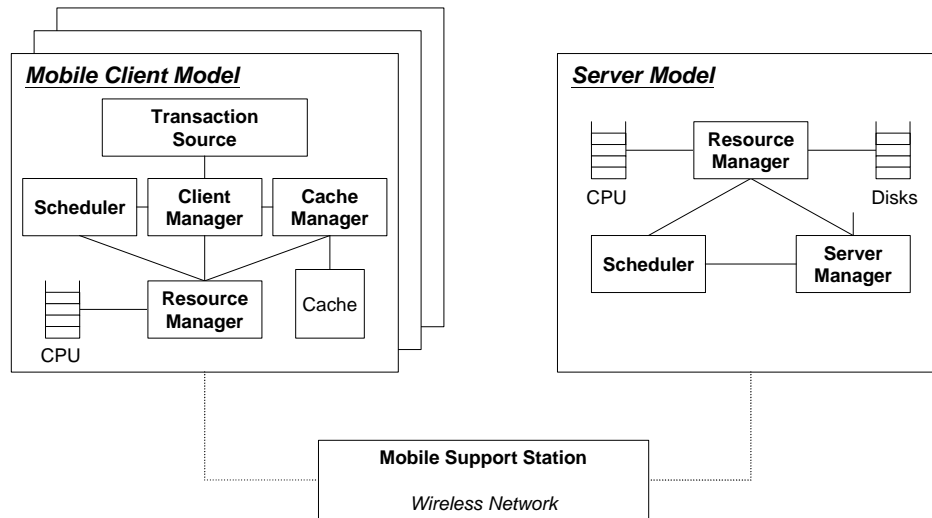
Figure 5: Mobile client-server database model.

5.1 Simulation Model

This section describes our simulation model. We can divide our simulation model into three parts: the database model, the transaction model, and the system model. The database model captures the characteristics of the database, such as the database size and object attributes. The transaction model captures the data object reference behavior for transactions in a workload. Finally, the system model captures the characteristics of the system's hardware and software.

Figure 5 shows the structure of our simulation model. It consists of components modeling the mobile clients and the database server that are connected via a wireless network. Each mobile client consists of a *cache manager* that uses the LRU (least recently used) cache replacement policy, a *scheduler* used as a concurrency control manager for read-only transactions, a *resource manager* providing CPU services and access to the wireless network, and a *client manager* coordinating the execution of transactions at the mobile client. Each mobile client also has a *transaction source*, which initiates client transactions according to the workloads described in the next subsection. If a transaction is aborted, it is resubmitted with the same data referencing string. The number of mobile clients is a parameter for the model. The *scheduler* on the server has the ability to store dependency information for data items, and the ability to verify update transactions. There is no transaction source module on the server, since all transactions originate on mobile clients.

19

5.1.1 Database Model

Database is composed of multiple data objects which have the same attributes. There are several attributes for each data object: the identifier, the timestamp, and the access probability indicator (hot-bound or cold-bound). The database model parameters are summarized in Table 1. The number of objects in the database, *NObject*, was chosen to be relatively small in order to experience data contention. For clients, the cache size, *CachePercent*, is a constant. The contents of each client's cache is fixed at the start of the simulation by uniformly choosing objects from the database. A portion of the entire database has a relatively high (low) probability of access by each mobile client, and this set of data items is referred to as the *HotBound* (*ColdBound*) data. The access probability for each set of data items will be specified later.

Table 1: Database Parameters.

Parameter	Meaning
NObject	Number of objects in the database
CachePercent	Percentage of cache size
HotBound	Data set with high probability of access
ColdBound	Data set with low probability of access

5.1.2 Transaction Model

The transaction model supports the following operations.

- *ReadObject*: Reads an object from its client's cache, if available. If the object does not exist in the cache, it reads it from the server database.

- *WriteObject*: Updates an object in the client's cache.

- *CommitTR*: Commits a transaction.

- *AbortTR*: Aborts a transaction.

A transaction is modeled as a finite loop of *ReadObject* and *WriteObject* operations, which are followed by either the *CommitTR* or the *AbortTR* operation. Table 2 summarizes the parameters that characterize a transaction. The number of *ReadObject* or *WriteObject* operations in a transaction is called the size of a transaction. Transaction size is uniformly distributed between *MinTRSize* and *MaxTRSize*. The parameter *HotAccessProb* (*ColdAccessProb*) presents the data access probability

20

for each mobile client. For both access ranges, the probabilities that an access to a data item in the range is a *WriteObject* operation are specified as *HotWriteProb* and *ColdWriteProb*, respectively. The delay parameters are exponentially distributed delay times used to model interactive systems.

Table 2: Transaction Parameters.

Parameter	Meaning
MaxTRSize	Maximum number of operations in a transaction
MinTRSize	Minimum number of operations in a transaction
HotAccessProb	Probability of access to *HotBound* data
ColdAccessProb	Probability of access to *ColdBound* data
HotWriteProb	Probability of *WriteObject* operation on *HotBound* data
ColdWriteProb	Probability of *WriteObject* operation on *ColdBound* data
ReadDelay	Average delay for a *ReadObject* operation
WriteDelay	Average delay for a *WriteObject* operation
TRState	State of transactions being processed

5.1.3 System Model

The system model consists of a network manager, clients, and server modules. The parameters for all these modules are summarized in Table 3.

Table 3: System Parameters.

Parameter	Meaning
NClient	Number of mobile clients
NetDelay	Average communication delay on the wireless network
DBAccessDelay	Average delay to access database
CacheAccessDelay	Average delay to access cache
DisconnectProb	Probability that a mobile client is disconnected from the server

Please note that *DisconnectProb*, called simply *Probability that a mobile client is disconnected from the server*, actually subsumes all instances of message losses suffered by mobile clients (including not only actual disconnections, but also cases of lost, misrouted, or erroneous packets).

5.2 Experiments

The main performance metric is system *throughput* measured in terms of committed transactions per second. Throughput depends on the particular settings chosen for the various physical system resource parameters. We present other performance measures: *average waiting time* and *the number of aborts* to provide additional insights into the fundamental tradeoffs for the proposed protocol under various workloads. Table 4 describes the settings for the parameters that are used to specify the system resources and overheads.

Table 4: Simulation Parameters.

Parameter	Value
NObject	400
CachePercent	10%
ReadDelay	10 ms
WriteDelay	40 ms
NClient	3, 6, 9, 12, 15
NetDelay	80 ms
DBAcessDelay	50 ms
CacheAccessDelay	10 ms
DisconnectProb	0.2

Our simulation model provides a simple yet flexible mechanism for describing client workloads. An access pattern can be specified separately for each client using the parameters shown in Table 5. A transaction is represented as a string of data access requests in which some accesses are for reads and others are for writes. The probability of a data access to the hot range is specified. The remaining accesses are directed to data items in the cold range. For both ranges, the probability that an access to a data item in the range will involve a write is specified.

Table 5 summarizes the workloads used in our experiments. Workload 1 has a per-client private hot region that is read and written by each mobile client and a shared cold region that is accessed by all mobile clients. This workload is intended to model an environment where users access and update their own private data, while retrieving some of the shared data items. Workload 2 has a relatively high degree of locality per mobile client and a moderate amount of sharing and data contention. Workload 3 is a low-locality and moderate-write-probability workload. It is used to examine the performance of the protocol in cases when exploiting dependency information is not expected to pay off significantly.

22

Table 5: Workload Parameters.

Parameter	Workload 1	Workload 2	Workload 3
MaxTRSize	5	5	8
MinTRSize	2	2	2
HotBounds	p to p+29, (p=15(n-1)+1)	p to p+59, (p=20(n-1)+1)	.
ColdBounds	241 to 400	rest of DB	all of DB
HotAccessProb	0.8	0.6	.
ColdAccessProb	0.2	0.4	1.0
HotWriteProb	0.3	0.3	.
ColdWriteProb	0.2	0.3	0.2

5.3   Results and Discussion

We run a number of simulations to evaluate the proposed protocol under the criteria of the number of aborts, the average waiting time, and throughput.

Number of aborts   Figure 6 and Figure 7 show the average number of aborts that occur before the protocol is able to commit a total of fifty transactions. When the window size is small, most of the transactions must restart several times before they are finally committed. For short window sizes, most read operations on cached copies cannot be locally verified by mobile clients. As a result, there is a possibility that a transaction that accessed consistent data and that does not produce a cycle is aborted.

As the window size gets larger, most of the accesses to cached copies can be verified using the dependency information, so the number of aborts is reduced. When the window is larger, aborts of transactions are due mainly to conflicts between transactions or inconsistencies in cached data caused by lost messages (e.g., due to disconnections or misrouted packets). As the window size gets larger, the protocol shows better performance under Workload 1, in which most of transactions at mobile clients access their own private data. Under this workload, the probability that $Dependent_{w,t}(x)$ contains dependency information for already read data items is very high. As a result, the mobile client can determine whether an active transaction might produce a cycle in the serialization graph or not. On the other hand, under Workload 3, more transactions are aborted compared with Workload 1. All mobile clients show a uniform access pattern to data items under this workload. As a result, $Dependent_{w,t}(x)$ for a requested data item x has dependency information for many data items from
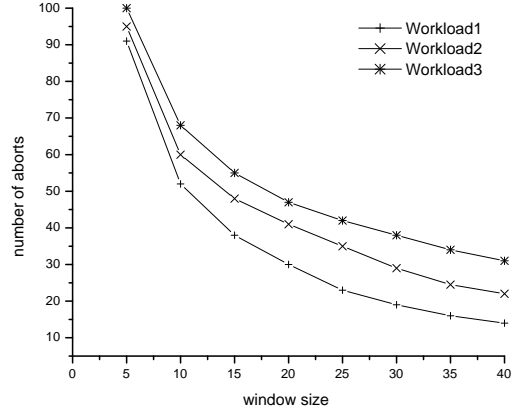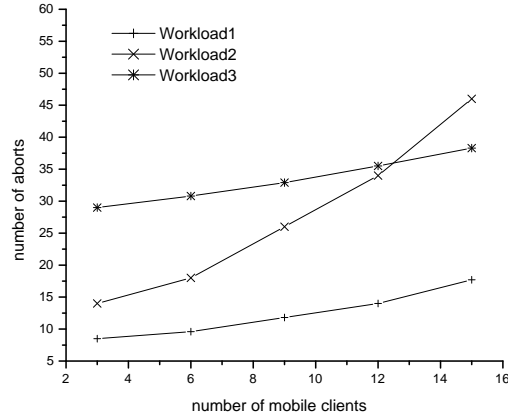
Figure 6: Number of aborts vs. window size.

Figure 7: Number of aborts vs. number of mobile clients.

the entire database. This makes the actual length covered by the list shorter.

In case of Workload 1, the number of aborted transactions is not reduced rapidly when the window size gets larger than approximately 30. This is because the number of hot-bound data items for each mobile client is limited. When the window size exceeds a certain threshold value, $Dependent_{w,t}(x)$ may include dependency information for most hot-bound data items for each mobile client. However, under Workload 3, since $Dependent_{w,t}(x)$ can not contain timestamps of all data items that show conflicts with x, some transactions can not be verified using this information. In the experiments, the protocol shows intermediate performance under Workload 2, since each mobile client has its hot-bound data items which are shared with some other mobile clients.

We also examined aborts for the cases when the number of mobile clients gradually increases from

24

3 to 15. As shown in Figure 7, the number of aborts increases slightly under Workload 1, due to the increased rate of conflicts on cold-bound data items. Accesses to hot-bound data items do not cause conflicts among transactions, since each mobile client has its own private hot-bound data set. Thus, under Workload 1, the number of aborts is less dependent upon the population of mobile clients in a cell. Under Workload 3, the number of aborts also increases slowly as the number of mobile clients gets larger. In this workload, each mobile client shows uniform access pattern for all data items in the database. Hence, most of data items are requested from the server, not from the cache. As a result, the increased population of mobile clients does not significantly change the probability that a transaction can be committed. However, under Workload 2, as the number of clients gets larger, more cold-bound data items are updated, which can make more cached copies of hot-bound data items stale. This increases the probability that the timestamp of an accessed data item x in the cache ($cache.x.ts$) is lower than $Dependent_{w,t}(y).x.ts$. Thus, for Workload 2 more transactions are aborted with the increasing number of mobile clients.

Average waiting time   In terms of waiting time, we measured the delay for a transaction until it can decide to either commit or abort. As shown in Figure 8, the protocol shows different patterns of waiting time under each workload. For Workload 1, transactions have to wait almost uniformly, approximately for 200 ms. This relatively low waiting time occurs for two reasons. First, most of the transactions are read-only. As a result, mobile clients can verify them autonomously, thus avoiding exchange of messages with the server, which saves time. Second, transactions access most data items from caches, which is very fast compared with requesting data from the server. On the other hand, for Workload 2, it takes much longer to decide whether to commit or abort a transaction, because the ratio of update transactions is relatively high when compared to the other two workloads. The waiting time gets longer with the increasing window size. This can be explained using Figure 6. When the window size is small, most of the transactions are autonomously aborted earlier, using $Dependent_{w,t}(x)$. As the window gets larger, such aborts are reduced. Hence, most of the update transactions completes, and they are sent to the server to be verified. In case of Workload 3, the delays are longer when compared to delays for Workload 1, since most of data items accessed by transactions must be acquired from the server. The slight increase of the waiting time for this workload is mainly due to the reduced number of earlier aborts.

We examined the impact of client population on the average waiting time. The result, shown in Figure 9, shows pattern similar to Figure 8. The waiting time for a transaction is relatively short under Workload 1, since we can expect that most of the hot-bound data items are maintained in each client's cache. In case of Workload 2, as the number of mobile clients gets larger, conflicts between
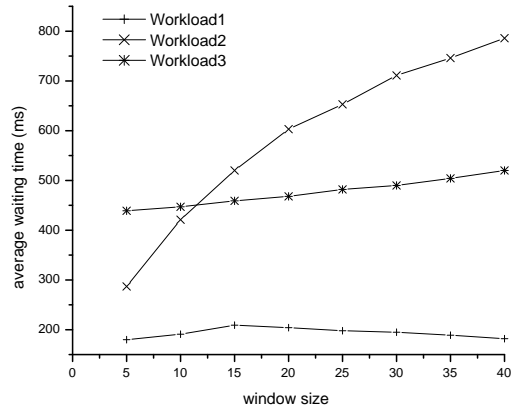
25

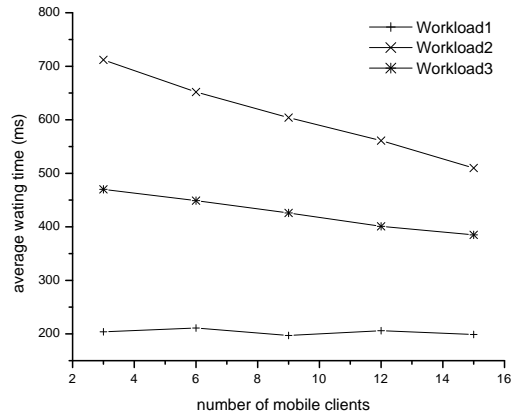Figure 8: Average waiting time vs. window size.

Figure 9: Average waiting time vs. number of mobile clients.

transactions increase. Some of these aborts of update transactions can be detected by mobile clients without sending the *commit_request* message and waiting for the decision of the server. As a result, the average waiting time decreases with the increasing number of mobile clients. For the same reason, the waiting time for Workload 3 decreases slowly.

Throughput Figure 10 and Figure 11 show the throughput results for the protocol under three workloads. Figure 10 presents the total system throughput for each workload as the window size is gradually increased from 5 to 40. In this experiment, the protocol provides the best performance under Workload 1. The protocol has a lower throughput for Workload 2, and the poorest performance for Workload 3. Due to no per-client locality in Workload 3, there is a relatively high possibility of
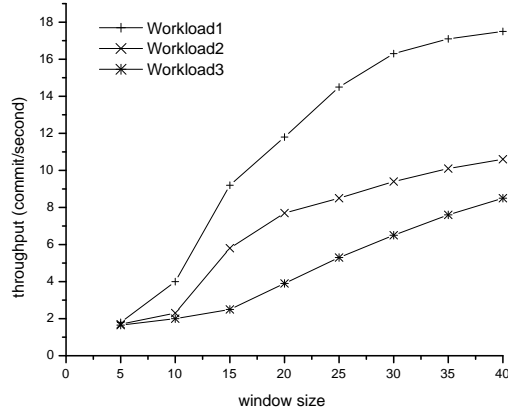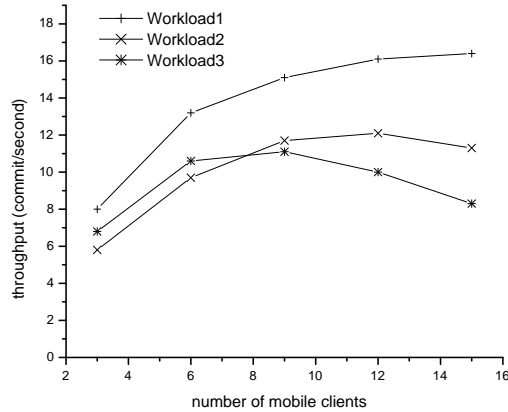
Figure 10: Throughput vs. window size.

Figure 11: Throughput vs. number of mobile clients.

data conflicts, thus in this case we caching is less beneficial than for the other two workloads. When the window size w for $Dependent_{w,t}(x)$ is very small, we cannot expect a satisfying performance for any workload, since most of the transactions have to be aborted when a mobile client receives $Dependent_{w,t}(x)$ attached to the requested data item x. As the window size gets larger, more transactions can be committed during the same period, due to the reduced number of aborts of transactions that cannot be verified by mobile clients (see Figure 6).

Finally, we examined the throughput of the proposed protocol varying the population of mobile clients from 3 to 15. The high degree of per-client locality and low conflict ratios between accessed data items let the protocol commit more transaction for Workload 1 when compared with the other two workloads. For read-only transactions, the mobile clients can verify more transactions, since

the dependency information delivered from the server may contain information for data items that are likely to be maintained in their own caches. For update transactions, more transaction can be committed since there might be fewer conflicts between them. The throughput of the protocol does not degrade for Workload 1, because the increased number of mobile clients does not cause more conflicts between transactions. However, for Workloads 2 and 3, throughput of the protocol degrades when the number of mobile clients exceeds approximately the value 10. This is due to the increased number of conflicts between transactions. For Workload 2, the increased number of write accesses to cold bound data items is the main factor which makes the cached data copies stale. As a result, more transactions must be aborted. For Workload 3, since all mobile clients show the same access patterns over the entire databases, the degree of conflicts is directly affected by the number of mobile clients.

6   Related Work

Several algorithms have been proposed to support concurrency control of transaction processing in mobile environments [2, 8, 9, 10, 11 14]. To support mobile transactions, the transaction processing models should accommodate the limitations of mobile computing such as low bandwidth communication, limited battery power, frequent disconnections, and reduced storage capacity [12]. Mobile computations should minimize aborts due to lost messages introduced, for example, by disconnections or misrouted packets. Operations on shared data must ensure correctness of transactions executed on both the server and mobile clients. A proper support for mobile transactions must provide for local autonomy, in order to allow transactions to be processed and committed on the mobile clients despite message losses.

Some local commit schemes for read-only transactions have been proposed in the literature. An algorithm proposed in [14] supports local commit of read-only transactions in server-initiated push-based data delivery environments. It uses a weaker notion of consistency, called *update consistency*, which still satisfies the requirement for mutual consistency of data. In this scheme, the server transmits control information, called control matrix, during each cycle. It helps mobile clients to determine whether read-only transactions read consistent values. In mobile clients, before a read operation is performed on a data item broadcast during a cycle, control information transmitted during that cycle is consulted to determine whether the read operation can proceed. Thus, if a transaction has not performed any write operation, it can be committed without any validation process involving the server. Another local commit scheme for read-only transactions is presented in [11]. This scheme was proposed for the Broadcast Disk environment [1], which is also a server-initiated data delivery

mechanism. To increase the number of read-only transactions that are successfully processed, the scheme proposes maintaining multiple versions of data items. By keeping older copies of data items for concurrency control purposes, the scheme can check the correctness of read-only transactions without any interaction with the server. Although this scheme can provide a considerable increase in concurrency, maintaining versions can be an overhead when data items are not updated frequently by the server. An algorithm presented in [9] can process read-only transactions locally without contacting the server. In this scheme, before a read operation is performed on a data item broadcast during a cycle, control information transmitted during this cycle is consulted to perform the *partial backward validation*. Hence, read-only mobile transactions can be processed locally by mobile clients, and only update transactions are sent to the server for the final validation.

As discussed above, the research studies undertaken to support local commit of read-only transactions use push-based data dissemination, in which data items are broadcast to mobile clients without any explicit request [1, 15]. Push-based data dissemination is an effective mechanism for large client populations, and avoids the limitations of the client-initiated schemes. In addition, the server is prevented from being jammed with multiple client requests. However, it is limited by the problem that it is difficult to predict accurately the needs of mobile clients. Sending irrelevant data results in a poor use of the channel bandwidth, and data might not reach the mobile clients in time. Read-only transactions can be validated more easily under the push-based data delivery, because all data items broadcast in a single cycle are guaranteed to be in a consistent state. Thus, read-only transactions which accessed data items delivered within the same period do not produce any cycles with other transactions. However, in pull-based data delivery, special mechanisms should be presented devised to allow for validation of read-only transactions by mobile clients, since we can guarantee neither the consistency among data items separately requested from the server, nor their consistency with cached copies. We have presented a concurrency control protocol in which mobile clients can validate read-only transactions autonomously without submitting them to the server.

7 Conclusions

We focused on transaction processing that increases the autonomy of clients in mobile database systems. We defined the dependency relation among updated data items. Lists of dependents send by the server to the mobile clients along with requested data items are used to build partial serialization graphs for each client. By receiving this dependency information, mobile clients can autonomously verify serializability of locally executed read-only transactions. This information can be used by

mobile clients to detect early the necessity for aborts of update transactions.

We have conducted simulations for various access patterns initiated by mobile clients in order to examine the performance of the proposed protocol. The performance of the protocol is heavily dependent on the depth of the dependency information (window size) for each data item. We found that the protocol exploits data access locality. In such cases dependency information may include more related data items that mobile clients need to autonomously verify read-only transactions.

Some enhancements can be made to the proposed scheme. We are now studying ways to prevent unnecessary aborts of read-only transactions in cases when the window size for dependency information is small. The number of read-only transactions that can not be verified by mobile clients is small when the window size exceeds a threshold value. These transactions can be submitted to the server instead of being aborted. We can make the size of dependency information uniform for all data items. To this end, a dynamic window size can be used so that the number of dependent data items included in the dependency information is fixed. With this strategy, we can expect that the response to every request for a data item will have the same size. We plan to investigate an autonomous protocol for the synchronous broadcast environments.

References

[1] S. Acharya, R. Alonso, M.J. Franklin and S.B. Zdonik, "Broadcast Disks: Data Management for Asymmetric Communications Environments," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp.199–210, 1995.

[2] D. Barbara, "Certification Reports: Supporting Transactions in Wireless Systems," in *Proceedings of IEEE International Conference on Distributed Computing Systems*, pp.466–473, 1997.

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

[4] B. Bhargava, "Concurrency Control in Database Systems," *IEEE Transactions on Knowledge and Data Engineering*, vol.11, no.1, pp.3–16, 1999.

[5] I. Chung, J. Ryu and C.-S. Hwang, "Efficient Cache Management Protocol Based on Data Locality in Mobile DBMSs," in *Current Issues in Databases and Information Systems, Proceedings of Conference on Advances in Databases and Information Systems, Lecture Note in Computer Science*, vol.1884, pp.51–64, Springer, 2000.

[6] J. Jing, A. Elmagarmid, A. Helal and A. Alonso, "Bit Sequences: An Adaptive Cache Invalidation Method in Mobile Client/Server Environments," *Mobile Networks and Applications*, vol.2, no.2, pp.115–127, 1997.

[7] A. Kahol, S. Khurana, S.K. Gupta and P.K. Srimani, "An Efficient Cache Maintenance Scheme for Mobile Environment," in *Proceedings of International Conference on Distributed Computing Systems*, pp.530–537, 2000.

[8] V.C.S. Lee and K.-W. Lam, "Optimistic Concurrency Control in Broadcast Environments: Looking Forward at the Server and Backward at the Clients," in *Proceedings of International Conference on Mobile Data Access, Lecture Note in Computer Science*, vol.1748, pp.97–106, Springer, 1999.

[9] S.K. Madria and B. Bhargava, "A Transaction Model to Improve Data Availability in Mobile Computing," *Distributed and Parallel Databases*, vol.10, no.2. pp.127–160, 2001

[10] E. Pitoura and B. Bhargava, "Data Consistency in Intermittently Connected Distributed Systems," *IEEE Transactions on Knowledge and Data Engineering*, vol.11, no.6, pp.896–915, 1999.

[11] E. Pitoura and P.K. Chrysanthis, "Exploiting Versions for Handling Updates in Broadcast Disks," in *Proceedings of International Conference on Very Large Databases* pp.114–125, 1999.

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

[13] M. Satyanarayanan, "Mobile Information Access," *IEEE Personal Communications*, vol.3, no.1, pp.26–33, 1996.

[14] J. Shanmugasundaram, A. Nithrakashyap and R. Sivasankaran, "Efficient Concurrency Control for Broadcast Environments," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, pp.85–96, 1999.

[15] K. Stathatos, N. Roussopoulos and J.S. Baras, "Adaptive Data Broadcast in Hybrid Networks," in *Proceedings of International Conference on Very Large Data Bases*, pp.326–335, 1997.