Distributed and Parallel Databases, 10, 127–160, 2001

A Transaction Model to Improve Data Availability in Mobile Computing

SANJAY KUMAR MADRIA madrias@UMR.edu
Department of Computer Science, University of Missouri-Rolla, MO, USA

BHARAT BHARGAVA bb@cs.purdue.edu
Department of Computer Sciences, Purdue University, West Lafayette, IN, USA

Recommended by: Jin Jing

Abstract. We incorporate a *prewrite* operation before a write operation in a mobile transaction to improve data availability. A prewrite operation does not update the state of a data object but only makes visible the future value that the data object will have after the final commit of the transaction. Once a transaction reads all the values and declares all the prewrites, it can *pre-commit* at mobile host (MH) (computer connected to unreliable mobile communication network). The remaining transaction's execution (writes on database) is shifted to the mobile service station (MSS) (computer connected to the reliable fixed network). Writes on database consume time and resources and are therefore shifted to MSS and delayed. This reduces wireless network traffic congestion. Since the responsibility of expensive part of the transaction's execution is shifted to the MSS, it also reduces the computing expenses at mobile host. A pre-committed transaction's prewrite values are made visible both at mobile and at fixed database servers before the final commit of the transaction. Thus, it increases data availability during frequent disconnection common in mobile computing. Since a pre-committed transaction does not abort, no undo recovery needs to be performed in our model. A mobile host needs to cache only prewrite values of the data objects which take less memory, transmission time, energy and can be transmitted over low bandwidth. We have analysed various possible schedules of running transactions concurrently both at mobile and fixed database servers. We have discussed the concurrency control algorithm for our transaction model and proved that the concurrent execution of our transaction processing model produces only serializable schedules. Our performance study shows that our model increases *throughput* and decreases *transaction-abort-ratio* in comparison to other lock based schemes. We have briefly discussed the recovery issues and implementation of our model.

Keywords: prewrite, mobile transaction, pre-commit, data availability

1. Introduction

Technological advances in cellular communications, wireless LAN and satellite services have led to the emergence of mobile computing, also called nomadic computing [12, 26]. As the technology advancing, millions of users carry portable computer and communicator devices that use a wireless connection to access world-wide global information network. Wide area and wireless computing suggest that there will be more competition for shared data since it provide users with ability to access information and services through wireless connections that can be retained even while the user is moving. Further, mobile users will have to share their data with others. The task of ensuring consistency of shared data becomes more difficult in mobile computing because of limitations of wireless communication channels and restrictions imposed due to mobility and portability [22]. Some of the problems involved in supporting transaction services and distributed data management in a mobile environment have been identified in [36]. The access to the information systems through mobile computers will be performed with the help of mobile transactions. However, a transaction in this environment is different than the transactions in the centralized or distributed databases in the following ways.

- The mobile transactions are long-lived transactions due to the mobility of both the data and users, and due to the frequent disconnection.
- The mobile transactions might have to split their computations into sets of operations, some of which execute on mobile host while others on MSS. A mobile transaction shares their states and partial results with other transactions due to disconnection and mobility.
- The mobile transactions require computations and communications to be supported by mobile service stations (MSS).
- As the mobile hosts move from one cell to another, the states of transaction, states of accessed data objects, and the location information also move.
- The mobile transactions should support and handle concurrency, recovery, disconnection and mutual consistency of the replicated data objects.

In general to support mobile computing, the transaction processing models should accommodate the limitations of mobile computing such as unreliable communication, limited battery life, low band-width communication and reduced storage capacity. Mobile computations should minimize transaction aborts due to disconnection. Operations on shared data must ensure correctness of transactions executed on both stationary hosts (computer connected to the fixed network) and mobile hosts. The blocking of a transaction's execution on either the stationary or mobile hosts must be minimized to reduce communication cost and to increase concurrency. Proper support for mobile transactions must provide for local autonomy to allow transactions to be processed and committed on the mobile host despite temporary disconnection.

Many concurrency control protocols [4] are based on the notion of locks where a data object in the database can be accessed only after a lock on that object has been acquired for the duration of the read or writes. That is, a read (write) operation can be executed only after a read (write)-lock is obtained. After acquiring the lock, the transaction executes its operation and then may release the locks. Since read operations do not conflict, many read transactions may share the read-lock on an object but sharing is not permitted if one of the locks is a write-lock. The above design is more suitable for database management systems that support short-duration transactions that read and write data objects for a short period of time. However, in case of long-duration transactions such as in mobile computing, the concurrency control algorithms based on above protocols suffer from performance degradation. Due to isolation requirements, the data items held by long transactions at mobile service station can not be released until the transaction commits. Therefore, once the transaction acquires a write-lock at mobile service station, the other transactions at mobile host may have to wait for very long before they get the lock. Thus, if short-duration transactions at mobile host want to read some data items held by a long-lived transaction at mobile service station, it will end up waiting for the long-lived transaction to commit. Also, if a transaction is considered as a basic unit of work, a significant amount of work may be lost in case of a failure. Therefore, it is desirable to make the response of a system fast particularly for read-only transactions. Also, the system should not delay short-duration transactions at mobile host due to the presence of long transactions at mobile service station.

Our objective in this paper is to discuss a mobile transaction processing model which increases data availability at mobile and stationary hosts, and at the same time can cope with problems discussed before. More specifically, our model supports both short and long transactions without blocking or aborting short-transactions. Our concurrency control protocols though use locks but it is optimistic in nature with no cascading aborts. Our system ensures serializability and ACID properties [6] for mobile database applications. We introduce a prewrite operation before a write operation in the transaction model. The operations in our mobile transaction processing model are *pre-read*, *read*, *prewrite*, *pre-commit* and *commit*. The semantics of these operations are explained below.

- Each mobile transaction has a prewrite operation before a write operation. A prewrite operation makes visible (the exact or abstract) the value the data object will have after the commit of the transaction.
- Once all the prewrites have been processed, the mobile transaction pre-commits at mobile host. A pre-committed transaction's results are made visible at mobile and stationary hosts before the final commit. This minimizes the blocking of other transactions to increase concurrency.
- The transaction continues its execution at mobile host by announcing prewrite values and then shifts the resource consuming part of the transaction's execution (updates of the database on disk) to the MSS. This reduces the computing cost on the mobile host.
- A pre-committed transaction is guaranteed to commit. This feature of our model avoids an undo or compensating transaction, which is costly in mobile computing.
- A pre-read returns a prewrite value whereas a read returns a write value.
- The transactions are serialized based on their pre-commit order. This feature of our model saves some computing cost as transactions which can not be serialized or deadlocked are aborted before pre-commit.
- Our model deals efficiently with the other constrained resources in mobile computing environment such as limited bandwidth and transmission time. For example, prewrite values are much smaller in size than the complete object and thus, they consume less bandwidth and transmission time.

Further to the discussion of the prewrite transaction model, we discuss the concurrency control algorithm, which control the concurrent execution of operations in our model. We have introduced one more type of lock (prewrite-lock) other than usual read- and write-locks. A prewrite-lock is converted to write-lock after pre-commit. Since two phase locking is widely implemented and accepted, our transaction model is implemented over that. However, only two phase locking is not sufficient to guarantee the correct execution in our model. Thus, we introduce one extra condition in order to ensure correct execution of concurrent transactions. This extra condition ensures that a transaction can not acquire any other lock after its lock-type is updated to another lock-type.

We discuss some interesting schedules that can occur in our transaction model. In particular, we show that a schedule with a prewrite operation is same as the schedule without a prewrite operation in case of simple data objects (i.e., a field value in a record). We formally discuss our transaction model and prove the correctness by showing that all possible schedules are serializable using serializability- based theorems. In addition, our simulation study shows that our concurrency control algorithm performs better in terms of *throughput* and *transaction-abort-ratio* with respect to other locking schemes [1, 10, 32]. Briefly, we have outlined the recovery issues and implementation of our model.

The rest of the paper is organized as follows. Section 2 outlines the mobile architecture and review some of the existing work in mobile transaction processing models. In Section 3, we discuss the transaction model using prewrites. Section 4 proposes a prewrite mobile transaction model. In Section 5, we discuss concurrent operations and locking algorithm. We discuss in Section 6 some serializable schedules. In Section 7, we discuss some recovery issues, summarise and analyse parameters which can affect our model and briefly discuss the implementation. We study the performance in Section 8. We conclude in Section 9.

2. Mobile architecture

In mobile database computing environment (see figure 1), the network consists of stationary or fixed hosts, mobile support stations (MSS) connected to stationary database servers and mobile hosts [12]. A mobile host (MH) changes its location and network connections while computations are being processed. While in motion, a mobile host retains its

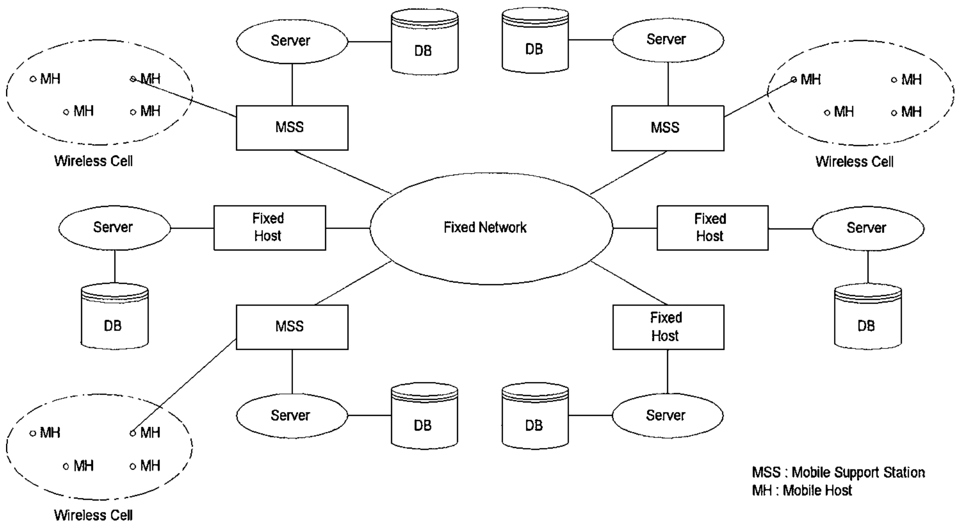


Figure 1. Mobile architecture.

network connections through the support of MSS with wireless connections. Each MSS has a coordinator which receives a transactions's operations from MH and monitor their execution in database servers within the fixed networks. These MSSs perform the transaction and data management functions with the help of transaction managers (TMs) and data managers (DMs) at database servers, respectively. Each MSS is responsible for all the mobile hosts within a given small geographical area, known as a cell. At any given instant, a MH communicates only with the MSS responsible for its cell. A MH may have some server capability to perform concurrency control and logging etc. Some MHs have very slow CPU and very little memory and thus, they act as an I/O device only. Within this mobile computing environment, shared data are expected to be stored and controlled by a number of database servers executing on MSS.

The database server support basic operations such as read, write, prewrite, preread, precommit, commit and abort operations. For a transaction's execution at MSS, the MH sends the transaction's operations to the coordinator at MSS, which sends them to the database servers within the fixed networks for execution. Similarly, the precommit and commit are handled by the coordinator at MSS. A transaction's operations may be submitted in multiple request messages. A submission requests may be for one operation or a group of operations. The subsequent requests are submitted once the previous steps are executed.

When a MH leaves a cell serviced by a MSS, a hand-off protocol is used to transfer the responsibility for mobile transaction and data support to the MSS of the new cell. This hand-off may involve establishing a new communication link. It involves the migration of in progress transactions and database states from one MSS to another.

In mobile computing, there are several possible modes of operations. The operation mode may be fully connected (normal connection), totally disconnected (it is not a failure of MH) or partially connected (weak connection). Also, mobile computers may enter an energy conservation mode, called doze state [27]. A doze state of MH does not imply the failure of the disconnected machine.

2.1. Review of research ideas in mobile transaction processing

The mobile transaction processing is an active area of research [18, 36]. We outline some of the existing ideas as follows.

- Semantic based transaction processing models [3, 31] have been extended for mobile computing in [33] to increase concurrency by exploiting commutative operations. These techniques require caching large portion of the database or maintain multiple copies of many data items. In [33], fragmentability of data objects has been used to facilitate semantic based transaction processing in mobile databases. The scheme fragments data objects. Each fragmented data object has to be cached independently and manipulated synchronously. This scheme works in the situations where the data objects can be fragmented like sets, aggregates, stacks and queues.
- In optimistic concurrency control based schemes [14], cached objects on mobile hosts can be updated without any co-ordination but the updates need to be propagated and validated at the database servers for the commitment of transactions. This scheme leads

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

- In pessimistic schemes, cached objects can be locked exclusively and mobile transactions can be committed locally. The pessimistic schemes lead to unnecessary transaction blocking since mobile hosts can not release any cached objects while it is disconnected. Existing caching methods attempt to cache the entire data objects or in some case the complete file. Caching of these potentially large objects over low bandwidth communication channels can result in wireless network congestion and high communication cost. The limited memory size of the MH allows only a small number of objects to be cached at any given time.
- Dynamic object clustering has been proposed in mobile computing in [27, 28] using *weak-read*, *weak-write*, *strict-read* and *strict-write*. Strict-read and strict-write have the same semantics as normal read and write operations invoked by transactions satisfying ACID properties [4]. A weak-read returns the value of a locally cached object written by a strict-write or a weak-write. A weak-write operation only updates a locally cached object, which might become permanent on cluster merging if the weak-write does not conflict with any strict-read or strict-write operation. The weak transactions use local and global commits. The "local commit" is same as our "pre-commit" and "global commit" is same as our "final commit" (see Section 3). However, a weak transaction after local commit can abort and is compensated. In our model, a pre-committed transaction does not abort and hence, requires no undo or compensation. A weak transaction's updates are visible to other weak transactions whereas prewrites are visible to all transactions. [15] presents a new transaction model using isolation-only transactions (IOT). IOTs do not provide failure atomicity and are similar to weak transactions of [27].
- An open nested transaction model has been proposed in [8] for modelling mobile transactions as a set of subtransactions. The model allows transactions to be executed on disconnection. It supports unilateral commitment of subtransactions and compensating transactions. However, not all the operations are compensated [8], and compensation is costly in mobile computing.
- A kangaroo transaction (KT) model was given in [9]. It incorporates the property that transactions in a mobile computing hop from a base station to another as the mobile unit moves. The mobility of the transaction model is captured by the use of split transaction [29]. A split transaction divides an on going transaction into serializable subtransactions. Earlier created subtransaction may commit and the second subtransaction can continue its execution. The mobile transaction is splitted when a hop occurs. The model captures the data behaviour of the mobile transaction using global and local transactions. The model also relies on compensating transaction in case a transaction aborts. Our model has the option of either using nested transactions or split transactions. However, the save point or split point of a transaction is explicitly defined by the use of pre-commit. This feature of the model allows the split point to occur in any of the cell. Unlike KT model, the earlier subtransaction after pre-commit can still continue its execution with the new subtransaction since their commit orders are based on pre-commit point in our model. Unlike KT, our model does not need any compensatory transaction.

- Transaction models for mobile computing that perform updates at mobile computers have been developed in [8, 27]. These efforts propose a new correctness criterion [8] that are weaker than the serializability. They can cope more efficiently with the restrictions of mobile and wireless communications. Our motivation is to increase availability and at the same time our model should remain within the correctness as defined by the classical serializability theory.
- In [16, 17, 19, 21], prewrite operations have been used in nested transaction environment to increase concurrency and to avoid undo or compensated operations. The notion of a recovery point subtransaction has been introduced. In a nested transaction tree, if a recovery-point subtransaction executed successfully, its effects are not to be discarded. In this paper, we exploit some of these ideas in order to increase reliability and availability in mobile computing environment.

3. Prewrite transaction model

In this section, we present the prewrite transaction model. We start the discussion of our model with some formal definitions. We illustrate the model with the help of examples. We briefly compare our prewrite transaction model with the transaction model handling multiversions of data.

Definition 1. A *prewrite operation* announces the value that the data object will have after the commit of the corresponding write operation. For example, a prewrite operation can announce the value in the following format [file-name, record number, <field-name, new value>] for a simple database application.

Definition 2. A transaction is called *pre-committed* if it has announced all the prewrites values and read all the required data objects, but the transaction has not been finally committed (updates on database are not performed).

Definition 3. A *pre-read* returns a prewrite value of the data object whereas a read returns the result of a write operation. A pre-read becomes a *weak-read* (in case the data objects involved are not simple) if it returns a prewrite value even though the transaction that announced the last prewrite has been finally committed. However, this weak-read should not to be aborted. This makes our weak-read different from the weak-read of [27]. Note that a transaction does not explicit contain a pre-read request. It is the system that returns a prewrite value in response, which we represent and call pre-read.

Let us consider two examples to show how a prewrite operation can be adopted to increase availability.

Example 1. Long-duration Transaction Application. Consider a construction company that exhibit a “model-house” (a toy-house) for the types of houses it has planned to construct. The “model-house” is shown to prospective customers. Consider a customer who has decided to buy a house based on pre-view of the “model-house”. We designate “house- construction” activity as house-construction transaction and “house-buying” activity as house-buying transaction. Since the “house-construction” and “house-buying” transactions are very long, the pre-view of “model-house” helps both the construction company and the customers in initiating other related transactions simultaneously. For example, once the customer has decided to buy a house, he may go to the bank and initiate a loan-application transaction after the pre-view of the “model-house”. His loan-application transaction is based on the pre-read of model-house and can commit independent of final commit of the “house-construction” transaction. The “model-house” corresponds to “prewrite version or value” for write operation of “house-construction” transaction and pre-view of “model-house” corresponds to pre-read operation of loan-application transaction.

Example 2. Data Structure Application. Consider the deletion of a record when physical pointers are in use in dynamic data structures consisting of records. If the space for the record were de-allocated as a part of deletion transaction, the problems may come if the transaction were to abort. For high concurrency, the storage allocation and de-allocation transactions need to continue once space for the record was de-allocated but before the transaction that de-allocated the space, is committed. As a result, the space may be allocated to other records, making it impossible for earlier transaction to re-obtain it in case it aborts. In case a new space is allocated for the record, the old pointers may no longer be valid. This problem can be avoided using notion of our pre-commit operation. Once the transaction announces the logical deletion (keep a pointer to the record being deleted), it can pre-commit. A pre-committed transaction does not abort, therefore, deallocated space can be used by others. The physical deletion of records (corresponds to final write) will be done after pre-commit. Thus, there is no need to acquire and keep the lock on the storage allocator until the transaction commits.

The transaction model using prewrite operations [17, 19] has the following features:

- Each prewrite operation makes visible the value the transaction will eventually write. Prewrites may have different semantics in different environments. For example, in case of simple data objects, the prewrite and write values may match exactly. For database files, the prewrites may only contain primary-key values and the new values of the fields of records. In case of design objects, prewrites may represent a model of the design. However, the final design released for manufacturing may differ from prewrite design of the model to some extent. For example, the final design may have a different colour shade than the prewrite design model. In case of a document object, the prewrite may represent an abstract of the detail document. For many applications, these small differences do not matter. In such an environment, the correctness criteria can be based on epsilon-serializability [30] which allows temporary and bounded inconsistencies in copies to be seen by queries during the period among the asynchronous updates of the various copies of a data item.
- Once the required data objects are read or pre-read and prewrites are computed, the transaction pre-commits. A transaction is required to read or pre-read all the required data objects before pre-commit because once a transaction releases a lock for a prewrite operation, it can not get a lock for read operation due to the condition of two phase locking [4]. After a pre-commit, the prewrites are made visible to other transactions

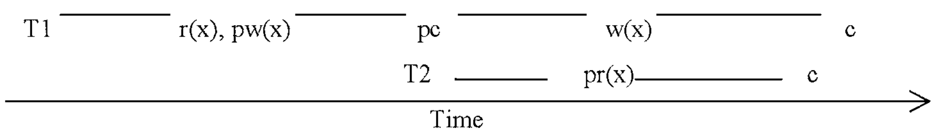


Figure 2. Two concurrent transactions.

for processing. Initially, prewrites are kept in the transaction's private workspace at the transaction manager level. Once the transaction pre-commits, they are posted in the prewrite-buffer. The data objects are always physically updated on the disk by the write operation. The prewrites are handled at the transaction manager (TM) level whereas physical writes are handled at the data manager (DM) level.

- The transactions commit order in the execution history of serializable transactions is decided at the time of pre-commit action. Thus, conflicts and deadlocks can be detected early.
- A transaction is not allowed to abort after the pre-commit. The prewrite operation provides non-strict execution without cascading aborts. In figure 2, T1 and T2 are two subtransactions where pw(x), w(x), pr(x), and r(x) are the prewrite, write, pre-read and read operations, respectively, for the data object x. Note that T2 pre-read the value written by T1. Also, other transactions can get write values of T1 before it commits. The transaction T2 commits before T1. In case T1 aborts after T2 is commited, there will not be a cascading abort. Since our model does not need "undo recovery" from transaction aborts, no compensating transaction is to be executed. In case the pre-committed transaction is forced to abort due to system dependent reasons such as system crash, the transaction restarts on system revival. To restart a failed pre-committed transaction from the last consistent state, prewrite and write logs are saved on stable storage [21].
- Our concurrency control algorithm is to be executed in two servers: For controlling pre-read (i.e., read of prewrite value) and prewrite operations at TM server, and read (i.e., read of write value) and write operations at DM server. Since prewrite values are made publicly visible after pre-commit, the lock-type held by the prewrite operation is converted to the lock-type for write operation after pre-commit provided no conflicting locks are held by other transactions. As before, the lock acquired for a prewrite operation is not released after pre-commit because the two phase locking [4] does not allow a transaction to acquire a lock after the transaction has released some locks.
- Our model relaxes the isolation property as the prewrites are made visible after pre-commit but before the final commit of the transaction. Also, durability of prewrites are guaranteed at the pre-commit point.
- The formal prewrite transaction processing algorithm is given below in figure 3.

3.1. Multiversions verses prewrites

Multiversions of data have been used for historical purposes as well as for issues related to the transaction management. Versions can substantially impact the level of concurrency.

```
1. A transaction T is submitted to the transaction manager (TM).
2. TM analyzes the transaction T to find out about its read and write requests.
   (/*pre-commit part of the algorithm executed at TM*/)
3. If the transaction T has read and write operations then
    Begin
    For all reads ∈ T
    Begin
    Send requests to DM for necessary locks ;
   DM obtains the necessary locks on the data objects provided no other transaction hold conflicting
   locks ; (*see Table 1 for operation compatibility matrix and Figure 6 for locking protocol*)
   Return values read to TM;   (/* read becomes pre-read if the value returned is a prewrite value*/)
   End;
   For all writes ∈ T
       Begin
           DM obtains the necessary locks for prewrites;
           Announce all the prewrite values at the workspace at TM;
            Store the necessary logs at DM;
            If T aborts, destroy T's workspace and announce abort of T;
       End;
         Begin (/*execute pre-commit*/)
        Update the lock-type of prewrite to lock-type of write provided no other transaction holds
        conflicting locks;
       (*see Table 1 for operation compatibility matrix and Figure 6 for locking protocol*)
        Write pre-commit log record;
        Announce pre-commit of the transaction;
        Release any lock-type held for read operations of the transaction;
        If aborts, submit T again;
       End;
   End;
4. (/* post pre-commit algorithm executed at DM*/)
     For each prewrite announced  at step 3
       Begin
       Update those data objects in the database for which prewrites have been announced;
       Store necessary log records;
       Release the lock-type held by the write operation of the transaction;
        Announce the "commit" of the transaction;
        If T aborts, it is resubmitted;
      End.
```

Figure 3. Prewrite transaction processing algorithm.

Table 1. Operation compatibility matrix.

	Pre-read	Read	Pre-write	Write
Pre-read	Yes	Yes	No	Yes
Read	Yes	Yes	No	No
Pre-write	No	Yes	No	Yes
Write	Yes	No	Yes	No

Many multiversion concurrency control algorithms [4, 11] use bounded number of versions for the data items to improve the performance of transaction processing. These schemes fall into two categories called mixed and pure multiversions. The mixed multiversions [2, 6, 7, 34] have two types of transactions, i.e., the read-only transaction and update transaction. The read only transactions read the old but consistent versions while update transactions manipulate only "current" version via two phase locking. However, the increase in the size and frequency of the updates limits the performance of the systems in case only "current" versions are available for their synchronization. Also, read-only transactions always read out-of-date data. This scheme may work well in mobile scenario if one allows only reads to occur at mobile host and writes at mobile service stations. Moreover, reads are also allowed to return out-of-date data, which however may not be accepted in many applications. Pure multiversions schemes using two phase locking [4, 13] utilizes the versions to allow the concurrent execution of the conflicting transactions. Since the concurrent access of the conflicting read-write actions is allowed on different versions of a data item in unrestricted fashion, the execution of a transaction must be validated before it can commit. In this case, the effort in executing the transaction that fails validation is wasted. This situation is undesirable in mobile computing as transactions are long. Pure multiversions are optimistic concurrency control schemes and aborts due to failed validation grows rapidly and therefore, performance becomes more prominent in mobile computing as the size of transaction grows.

In our prewrite transaction model, we have two versions of a data item; prewrite and write. However, the prewrite version does not represent the previous or old version of the current data version (write value), but it represents either a copy of the current version (exact) or the abstract value of the future write version. Reads in our model returns either the current version or an abstract value of the current version, which is not an old value. Therefore, the prewrite version or value in our model is different from the version concept in multiversions schemes. In our model, there is no validation phase in order to commit a transaction. Instead, we have introduced another phase which we call precommit, after which a transaction does not abort (if it is forced-abort, then it restarts). The main idea is that with the given mobile computing constraints, a transaction model has to be balanced to cope with limited resources but at the same time enhance availability. We also want to maintain the autonomy of the mobile host with respect to read and write operations.

[24] presents an optimistic multiversion concurrency control mechanism where multiversions are applied to the concepts of commutativity introduced by Weihl [35]. In our model,

we show that a general commutativity exists between a pre-read and write operation in case data object is simple.

4. Mobile transaction processing with prewrites

In this section, we see how prewrites can be adapted in the mobile transaction processing environment to improve availability. We discuss two cases with examples.

Case 1: MH has limited server capability to do some transaction processing, logging, and to execute pre-commits. Here we consider that mobile hosts have a high speed CPU along with some disk storage capabilities. This enables us to place some data locally at MH and manage some part of the transaction execution at MH. The data at MH are accessed by the mobile applications. Lock requests on the data objects are obtained by a request to MSS before processing.

In our mobile transaction model, a transaction begins its execution at mobile host (MH). When a transaction arrives at MH, the transaction's read requests are processed at MH with the consistent cached prewrite values of the data objects. For remaining reads, for which the MH has no consistent cached prewrite values of the data objects, the MH sends request to the MSS. When a read-only transaction arrives at MSS, it returns the prewrite values in response. If prewrite versions are not available at the server, the write values are returned. In case the values returned are prewrite values at MSS, the MSS identifies each value as a prewrite version. In case the transaction needs the version after the write, it has to initiate a read again. Once all the requested reads are processed by MSS and received by MH, the transaction declares all the prewrite values for the data objects at MH and pre-commits. A pre-committed transaction's execution is then shifted from mobile host to the stationary database servers for the completion of its remaining execution. This moves the expensive part of the mobile transaction execution to the static network. At the stationary host, the transaction makes the disk updates for all the data objects for which the prewrite values and pre-commit have been declared earlier and then it commits (see figure 4). Thus, the prewrite values of the data objects are made visible by MSS to other MHs and MSSs before those data objects are finally updated at the stationary host. This increases the concurrency among parallel running transactions at various MHs and MSSs. Prewrites are stored in the transaction's private workspace at MH and on pre-commit, are moved to MH's disk as well as to the MSS. Once updates have been made at MSS, the transaction finally commits at MSS. Note that once the transaction pre-commits, it does not abort. A transaction's commit is then communicated to MH. On receiving the commit acknowledgement, MH can clear all its logs. The formal mobile transaction-processing algorithm is given in figure 5 after the discussion on concurrent operations and locking.

Example 3. Real-time Application. Consider a newspaper reporter who is travelling in his van equipped with a mobile computer (thus, acts as a mobile host). Suppose while travelling

Start——— Reads/Prewrites ——— Pre-commit ——— Writes ——— Commit

Part of transaction executed at MH   Part of transaction executed at MSS

Figure 4. Transaction processing in mobile computing.

```
1. A transaction T is submitted to the mobile host (MH).
2. Mobile host's TM analyzes the transaction T to find out about its read and write requests (we
assume here that MH has some server capability).
3. It redefines the transaction T by converting request for reads to pre-reads and writes to prewrites.
   (/*pre-commit part of the transaction executed at MH*/)
4. If the transaction T has pre-read, read and write operations then
  Begin
    For all pre-reads and reads, writes ∈ T
   If DM at MH currently do not hold requested locks on the data objects or have old prewrite
versions then the TM at MH sends a request to MSS to acquire pre-read-, read- and prewrite-locks on
the data objects which MH currently do not hold;
After DM at MSS acquires the necessary locks for MH, it returns prewrite (if any) and write values to
DM of MH;  (Only those pre-write and write values that are not available locally at MH. However, in
that case also locks need to acquire to make sure that others do not hold conflicting locks at MSS)
For all writes ∈ T
       Begin
         If T has no blind-writes then
          MH announces all the prewrite values at MH;
          Store the prewrite logs at MH;
          Write pre-commit log record, and destination move log record if moving to other cell;
          Send prewrite values, prewrite logs and other log records to MSS;
         Begin
         MSS accepts the prewrite values and logs and send acknowledgement to MH;
         MSS update the prewrite-lock to write-lock provided no other transaction
         holds conflicting locks (*see Figure 3*);
         End;
          Else
         Begin
         If T has blind-writes then it requests MSS to acquire all prewrite-locks for all the data
objects
         for which prewrite values have been announced earlier;
        (*MH need not wait for step 4 in case transaction has blind-writes*)
        If prewrite-locks are not acquired, send message to MH to discard the prewrite values
        announced and corresponding logs;
        End;
 End;
      End;
4. For each prewrite announced      (/*post pre-commit algorithm executed at MSS*/)
      Begin
        Update those data objects in the database for which prewrites have been
        announced;
         Store necessary log records;
        Send MH the write-value; (*This will help in avoiding reading prewrite values at MH*)
      End;
```

Figure 5. Mobile transaction processing algorithm.

he encounters an accident site which he would like to report immediately. He does this by initiating a reporting transaction that consists of a "headline-report" about the accident. He sends it to the newspaper office (acts as a MSS). The newspaper office immediately displays the headline-report on their web page as well as on electronic bulletin board. The reporter at the first instance did not prepare and send the full report due to the following reasons. At the time of reporting, the wireless connection might be weak, or he noticed the wireless network congestion or his mobile computer might be running out of battery power. Headline-report reduces the blocking of other transactions at stationary host (MSS), as it may suffice the requirement of many other related transactions. For example, based on headline-report, some transactions like reservation of beds in hospitals for victims, and transactions for seat reservations in airlines for the relatives to travel to the accident site can be initiated. Also, once the headline-report arrived at MSS, the reporting transaction's remaining execution is shifted to the MSS to file the complete report. There are two possibilities to complete the report. The newspaper office (MSS) may contact some other reporter in that area to go to the accident site physically and complete the report. The reporter with mobile computer can also prepare the complete report at the time of disconnection (i.e., lunch time) or during weak connections. Thus, MSS can deal effectively with the situations like failure of MH or in case the mobile host has crossed the cell or is in doze mode. At the time of reconnection, the complete report would be incorporated into the database. The "headline-report" can also to be transmitted to other base stations in the fixed network so that transactions there can also be executed without delay.

In context of our transaction model, the reporting transaction has two parts. The first part consists of "headline-report" and the second part is the final "detailed-report". The headline-report corresponds to the prewrite version of our transaction model and the "detailed-report" is the final write. Once the report reaches at MSS, it denotes the completion of pre-commit of the reporting-transaction. Once the headline-report pre-committed, the reporter can not contradict it. However, he can make some changes such as casualty figures etc. This is in accordance to our notion of pre-commit. The benefits of enforcing pre-commit is that in such real-time situations transactions rarely abort (i.e., headline-report was a hoax).

Case 2: MH has very slow CPU and small memory, thus acts as an I/O device only. Due to weight and size restrictions, the MH may have few resources. For example, very slow CPU and small memory. In this scenario, MH can not do any major computation and rely on static MSS. The MSS executes the steps and sends the result to the MH. The main disadvantage is that the transaction execution is possible only when MH is connected to the MSS. Due to the slow communication and low bandwidth, the response time of the operations of the transactions is more.

In case the mobile host can not execute the transaction at MH (i.e., MH only acts as I/O), it can submit the transaction to the MSS. The server at MSS returns all the required values and declares all the prewrite values at the database servers in the fixed network corresponding to the write operations. After the transaction pre-commits, the prewrite values are send to the mobile host and at the same time, the rest of the transaction starts executing at the MSS. Once the transaction commits finally, it reports back to MH. In this case MH only acts as client with no processing capability.

Example 4. Stock Buying-Selling Application. Consider a stock-selling transaction that is initiated by a mobile user at MH and submitted to MSS. Suppose this transaction is accepted at MSS for selling stock based on approximately at average price a day earlier. The prewrite value here represents the approximate amount of money available based on the average price of the stock a day earlier. This information is made available to the MH before the stock-selling transaction is committed at MSS on the current price. Once this information is received at MH, the user can initiate another stock-buying transaction based on prewrite value of the approximate amount of money available. He can send his "buying-order" to the corresponding MSS before he leaves that cell. Thus, he can execute both buying and selling transactions together.

4.1. Mobile transaction model and partially replicated system

In case of partially replicated data objects, the transaction can announce the prewrite values of those data objects available in its current cell and pre-commits in the current cell. It does not wait to announce the prewrite values of all other required data objects by moving into different cells as it will block other transactions until it visits those cells. However, our approach of pre-committing in the current cell in such situations can also create some problems. For example, once a transaction is pre-committed, it can not again announce prewrite values for the objects in other cells. This is due to the fact that once the prewrite values are made visible at MSS by releasing some locks, it can not again acquire locks due to the two phase locking. This problem can be resolved in the following two ways:

a. Either the transaction requests all the locks for all the required data objects and wait until all the locks are granted. This strategy will delay the execution severely in case some links are down.
b. The other approach to solve the problem is by using either the nested transaction [23] or split transaction [29] as follows:

- **Nested Transaction Approach:** To deal with the problem stated above, we use the nested transaction model [23]. Once a transaction has announced all the prewrite values for the data objects available in its current cell and pre-committed, a new subtransaction is created. The earlier subtransaction is serialized before this new subtransaction based on pre-commit point. The earlier subtransaction's execution can be continued at the old MSS. The new subtransaction's responsibility is shifted to the MSS of the new cell with the help of hand-off protocol. Thus, both the transactions can be executed independently and concurrently in their respective cells. The pre-commit points are the save points of the nested transaction execution. Thus, they provide failure-tolerant execution. In case the earlier transaction aborts in the previous cell, its own effects and the new subtransaction are not undone. The aborted transaction can be restarted from the last pre-commit point.
- **Split Transaction Approach:** Another way of handling partially replicated data objects is to split [29] the transaction as soon as the MH moves to a new cell. The splitted transaction acts as a new transaction and therefore, can continue its processing in the new cell.

Prewrites help in partial replication of the data objects by creating prewrite versions of some of the data objects in the cells it moves. That is, it can make some new servers to support its files. For example, suppose a transaction at MH has send a “headline-report” about the accident and moved to another cell. In the new cell, it sends the “headline-report” to its new MSS also. The MSS can, therefore, get the headline-report before the complete report is processed at earlier MSS. Thus, the new MSS can serve those mobile transactions which require headline-report for further processing like reservation of beds in hospitals, in the current cell. Thus, the transactions at the new MSS are not blocked until the completion of earlier transaction at the previous MSS.

5. Concurrent operations

Transactions in our model are executed concurrently. To avoid concurrency anomalies, their execution are controlled so that final result of the execution is equivalent to the result of some serial execution of the transactions. Concurrency control algorithms are divided into optimistic and pessimistic [4]. Pessimistic approach block transactions by deferring the execution of some conflicting transactions whereas optimistic algorithms instead of blocking transactions they validate their correctness at commit time. Though optimistic algorithm seems to be useful in mobile computing environment, but more transactions are aborted and possibly needed restart which is costly as it consumes more resources limited in such an environment. Our concurrency control algorithm though uses locks but behave like optimistic in the sense that read values are made available before the final commit of transactions which updates those values.

In mobile transaction model, a pre-read operation can be executed at both MH and MSS at the same time. A pre-read can be executed at MH (MSS) while a write can take place at MSS (MH) concurrently. Similarly, a read at MSS can be executed concurrently with a prewrite at MH. A prewrite operation at MH can also be executed concurrently with another write operation at MSS since prewrites are managed at the transaction manager level at MH whereas the writes are performed at the data manager level at MSS. The operation-compatibility matrix of the various operations is given in Table 1. However, there are some interesting cases as follows:

Case 1: Suppose a pre-read is currently being executed at MH and at the same time, the transaction that has announced the prewrite values finally commits at MSS (final updates are performed) (see figure 6(a)). In this case, the pre-read will return a prewrite value that might be different than the last write value. For example, if the data object x is the simple data object, the read transaction T2 can commit before T1 as the prewrite and write values will be same. In case x is a design object, the system designate the read of T2 as a weak-read since the final design may differ from the prewrite value. The transaction T2 can resubmit its read request later to MSS if it needs the latest complete model of the design.

Case 2: In Table 1, observe that a read operation is compatible with a prewrite. Consider a case where a read transaction commits at MH after the transaction that announced the prewrite operation, has been pre-committed. The read in this situation will return an old

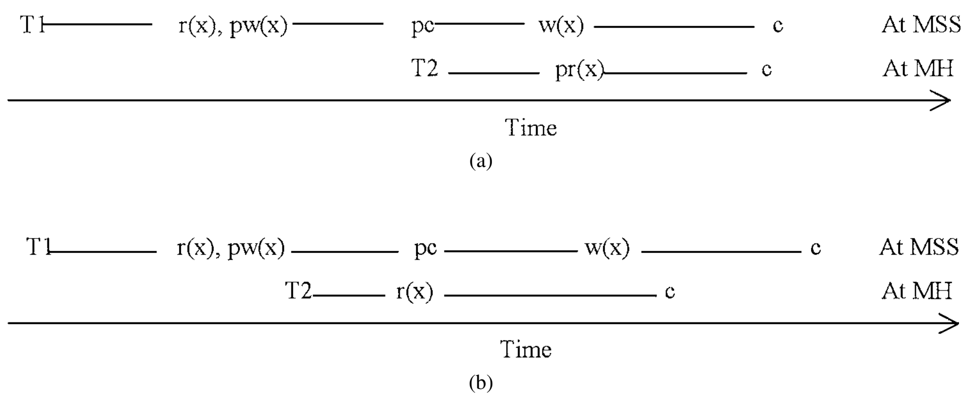


Figure 6. (a) The situation in Case 1. (b) The situation in Case 2.

value. However, this is not a significant problem because the transaction can still be serialized (see Section 6). For example, transaction T2 returns a write value, however it commits after T1 has pre-committed. The transaction T2 can be serialized before T1.

5.1. Concurrency control and locking

In this section, we develop a concurrency control algorithm to control the conflicting operations in our mobile transaction model. The concurrency control algorithm is executed in two phases and handled at MSS. In the first phase, the concurrency control for controlling prewrite and pre-read operations is performed at the transaction manager level at MSS. In the second phase, the concurrency control for controlling write and read operations (to access write values) is performed at the data manager (DM) level and also at MSS since the data managers are accessed only while performing updates on the databases.

We use read-lock for read, pre-read-lock for pre-read, prewrite-lock for prewrite, and write-lock for write operations, respectively. A prewrite-lock conflicts with other pre-read- and prewrite-locks, however, it does not conflict with read- and write-locks. A prewrite-lock can not be released after pre-commit as the transaction has to still get a write-lock for final updates. A prewrite-lock acquired by a pre-committed transaction is converted to a write-lock provided no other transaction holds the conflicting locks. Once a prewrite-lock is updated to a write-lock, the same transaction can not acquire any other lock. However, a pre-read-lock can be acquired by others in order to access prewrite values. There will be no a deadlock involving the transactions which are pre-committed. This is due to the fact that prewrite- and write-locks are acquired in an ordered fashion so deadlocks will occur only at the time of acquiring prewrite- or read-locks. Thus, a pre-committed transaction will not be aborted due to the deadlocks. The formal locking protocol is given in figure 7.

Locks on the data are managed and provided by the MSS. In case of replicated objects, the number of locks to be acquired in our algorithm depends on the particular replication algorithm used. The two main replication algorithms used are majority consensus and

Pre-Read-Lock(X): Grant the requested pre-read-lock to a transaction T on X if no other transaction holds a prewrite-lock on X.

Read-Lock(X): Grant the requested read-lock to a transaction T on X if no other transaction holds a write-lock on X.

Prewrite-Lock(X): Grant the prewrite-lock to a transaction T on X if no other transaction holds a Prewrite- or pre-read-lock on X.

Write-Lock(X): update a prewrite-lock on X held by a transaction T to write-lock if

```
Begin
If the write-lock-wait queue for X is empty then
  Begin
   If the transaction T is pre-committed and no other transaction holds a read- or
   write-lock on X then
   convert prewrite-lock to write-lock;
  End;
    else
     Begin
     put the transaction T in a write-lock-wait queue for X;
     End;
End.
```

Figure 7. The locking protocol.

read-one-write-all (ROWA) [4]. In majority consensus algorithm, locks are acquired on majority of sites whereas read-one-write-all (ROWA) requires lock on all the sites. In case read/write ratio is less, ROWA is preferred otherwise majority consensus will be preferred.

6. Serializable schedules in mobile transaction model

In this section, we formalize our mobile transaction model and discuss various possible schedules that can occur during an execution history of transactions.

A read operation executed by a transaction T_i on a data object x is denoted as $r_i(x)$, write as $w_i(x)$, pre-read as $pr_i(x)$ and prewrite as $pw_i(x)$. The pre-commit is denoted as pc_i, commit as c_i and an abort as a_i. When a transaction commits, its changes are applied to the database. If it aborts before pre-commit, the changes are discarded. A transaction is correct if it maps the database from one consistent state to another consistent state. Formally, a transaction T_i is a partial order with ordering $<_i$ where

1. $T_i \subseteq \{ pr_i$(x), r_i(x), pw_i(x), w_i(x) | x is a design object$\} \cup \{ pc_i,\ c_i,\ a_i\}$.
2. If $a_i \in T_i$ if and only if $pc_i \notin T_i$ and $c_i \notin T_i$.
3. if t is c_i or a_i then for any other operation p $\in T_i$, p $<_i$ t and if t is pc_i, then $pc_i <_i c_i$.
4. if pr_i(x), r_i(x), pw_i(x), w_i(x) $\in$ T then either pr_i(x)$<_i$ w_i(x), or w_i(x)$<_i$ r_i(x), or r_i(x) $<_i$ w_i(x), pw_i(x) $<_i$ pr_i(x) or pw_i(x) $<_i$ w_i (x), or pr_i(x) $<_i$ r_i (x).

Consider a set of transactions in $\mathbf{T} = (T_1,\ T_2, \ldots,\ T_n)$ which are modelled by a structure called a history. Formally [4], a history H over T is a partial order $(\Sigma,\ <_n)$ where

1. H $= \bigcup_{i=1}^{n} T_i$;
2. $<_H \supseteq \bigcup_{i=1}^{n}\ <_i$; and
3. for any two conflicting operations p and q either $p <_H\ q$ or $q\ <_H\ p$.

A serial history is one where for every pair of transactions T_i and T_j either all the operations executed by T_i precede all the operations executed by T_j or vice-versa. Two histories H and H′ are equivalent if they are defined over the same set of transactions and if p and q conflict and $p <_H q$ then $p <_H q$. A history is conflict serializable [4] if it is equivalent to some serial history. To determine whether a given history is conflict serializable, we analyze the graph derived from the history called serialization graph. The serialization graph of H, denoted by SG(H), is a directed graph whose nodes are transactions in T and has edges $T_i \rightarrow T_j$ if one of the T_i's operations precede and conflict with one of the T_j' s operation. A history H is conflict serializable if and only if SG(H) is acyclic [4]. An execution history will include lock and unlock operations. Each operation p_i(x) is preceded by a lock operation pl_i (x) and followed by an unlock operation $pul_i(x)$. Also in the history, $pwl_i(x) \rightarrow wl_i(x)$ implies that a prewrite-lock obtained by transaction T_i on object x is converted to a write-lock.

Now consider the following case histories to explain the various situations that might occur during concurrent execution of transactions.

Case 1: In this case we consider simple data objects and see that a history with a prewrite is same as the history without a prewrite. Consider the following history H:

$$pwl_1(x)pw_1(x)rl_2(x)r_2(x)rul_2(x)c_2pc_1(pwl_i(x) \\ \rightarrow wl_1(x))prl_3(x)pr_3(x)\ prul_3(x)c_3w_1(x)wul_1(x)c_1$$

The serialization graph of the above history will have an edge $T_1 \rightarrow T_3$ due to conflicting operations $pw_1(x)$ and $pr_3(x)$ and an another edge $T_2 \rightarrow T_1$ due to conflicting operations $r_2(x)$ (denotes the reading of the initial write value of x) and $w_1(x)$. Therefore, the above history is conflict serializable in the order $T_2 \rightarrow T_1 \rightarrow T_3$. It is equal to a serial schedule $T_2T_1T_3$. However, the above history is not strict since T_3 reads the value from T_1 but commits before T_1. Note that according to our algorithm, a transaction T_1 can not abort after its pre-commit operation is executed. In case of its failure after pre-commit, the transaction T_1 will restart from the failure point. Therefore, it will not introduce cascading aborts [4]. Hence, it is safe to allow non-strict executions in our algorithm. We observe that the above history has commutative operations $pw_1(x)$ and $r_2(x)$. That is, $pw_1(x)r_2(x)$ is same $r_2(x)pw_1(x)$. This is because a read returns the value from the write-buffer and therefore, a read before or after a prewrite will return the same value. We note that pre-read and write operations are also commutative in case the data objects involved are simple. In this case, a pre-read after a prewrite or a read after its associated final write will return the same value. However, in case of design objects, after the write, a pre-read may not be same as read since a write value (final value) may differ from its prewrite value (working copy). After taking into account these commutative operations, the above history will be equivalent to:

$$rl_2(x)r_2(x)rul_2(x)c_2pwl_1(x)pw_1(x)pc_1(pwl_1(x) \\ \rightarrow wl_1(x))prl_3(x)pr_3(x)\ prul_3(x)c_3w_1(x)wul_1(x)c_1$$

This history provides a serial history $T_2T_1T_3$. If we remove the prewrite operations from the above history, it will be a serial history consisting of only read and write operations. Thus, we have shown that in the case of simple data objects, any history with prewrites will be equivalent to a history without prewrites. That is, any system which permits pre-read, read, prewrite and write is same as the system with read and write operations in the sense that reads in both the systems will return the same value. However, the system with prewrites permits more concurrency than the system with normal read and writes.

Consider another history H':

$$pwl_1(x)pw_1(x)rl_2(x)r_2(x)rul_2(x)c_2pc_1(pwl_1(x) \\ \rightarrow wl_1(x))pwl_3(x)pw_3(x)pc_3w_1(x)\ wul_1(x)c_1(pwl_3(x) \\ \rightarrow wl_3(x))w_3(x)wul_3(x)c_3$$

The serialization graph of the above history has an edge $T_1 \rightarrow T_3$ due to conflicting prewrite and write operations. The graph also has an edge $T_2 \rightarrow T_1$ due to coflicting read and write operations. Therefore, the above history is conflict serializable in the order $T_2 \rightarrow T_1 \rightarrow T_3$. Observe that the ordering $T_1 \rightarrow T_3$ for two prewrite and write operations are same. This is required so that the prewrites and writes of two different transactions are performed in the same relative order. Also, the transaction T_3 sets the prewrite-lock after T_1 has converted its prewrite-lock to write-lock. Since two write-locks conflict, and also T_1 completes its prewrite before T_3's prewrite, therefore, T_3 has to wait until T_1 completes its final write.

Case 2: In this case we see that once a transaction's prewrite-lock is updated to the write-lock, it can not acquire any other lock. Consider the following history:

$$pwl_1(x)pw_1(x)pc_1(pwl_1(x) \\ \rightarrow wl_1(x))prl_2(x)pr_2(x)pwl_2(y)pw_2(y)pc_2\ (pwl_2(y) \\ \rightarrow wl_2(y))w_2(y)prul_2(x)wul_2(x)c_2w_1(x)rl_1(y)r_1(y)rul_1(y)wul_1(x)c_1$$

The above history is not conflict serializable though it satisfies our locking protocols and two phase locking. The schedule has a cycle since T_1 precedes T_2 on x and T_2 precedes T_1 on y. Additional restrictions must be introduced to disallow such an execution. Therefore, if a transaction's prewrite-lock is updated to a write-lock then the transaction can not acquire any other lock on any data object.

When this rule is introduced in the above history, T_1 can not acquire the lock on the data object y, therefore, there will not be a cycle.

Case 3: In this case, we see that a prewrite-lock can not be updated to a write-lock if some other transaction is holding a conflicting lock. Consider a partial history: $rl_1(x)r_1(x)pwl_1(x)$ $pw_1(x)rl_2(x)r_2(x)pc_1(pwl_1(x) \rightarrow wl_1(x))$

The above history is not allowed as T_2 holds a read-lock on x and conflicts with the write-lock acquired by T_1 after pre-commit. Thus, the prewrite-lock is updated to write-lock only if there is no conflicting lock (see write-lock rules in figure 7).

Consider another partial history:

$$pwl_1(x)pw_1(x)pc_1(pwl_1(x) \rightarrow wl_1(x))plw_2(x)pw_2(x)w_1(x)pc_2(pwl_2(x) \rightarrow w_2(x))$$

The above history is not allowed as T_1 and T_2 now hold conflicting locks. However, if $wul_1(x)$ appears before pc_2 then the history is allowed.

Case 4: In this case, we see that a transaction, which returns an old value, can be serialized in the history. Consider another history:

$$rl_1(x)r_1(x)pwl_1(x)pw_1(x)rl_2(x)r_2(x)pc_1rul_2(x)c_2(pwl_1(x) \rightarrow wl_1(x))$$

The above history is allowed. T returns an old-value since T_1 has been pre-committed before T_2 commits. However, this history can be serialized if we move operations of T_2 before T_1.

Note that our serializabile history satisfies the Q-class [25] of serializable history.

6.1. Proof of correctness

In this section, we formally prove the correctness of our algorithm and the locking protocols. We prove that the schedule produced by our locking protocols is conflict serializable. Our approach is based on the standard way of proving correctness of transaction processing algorithms.

Here, we state some properties based on our locking protocol as follows:

Property 1. *If o is an operation then $ol(x) < o(x) < ou(x)$.*

From two phase locking rule, we have the following property:

Property 2. *If $p_i(x)$ and $q_i(y)$ are two operations under T_i then $pl_i(x) < qul_i(y)$, i.e., for all lock operations $l_i \in T_i$ and un-lock operations $ul_i \in T_i$, $l_i < ul_i$.*

From the lock-upgrading rule, we have the following property :

Property 3. *If* $(pwl_i(x) \to wl_i(x)) \in T_i$ *then*

1. *for any operation* $ol_i \in T_i$, $ol_i(x) <_i (pwl_i(x) \to wl_i(x))$. *That is, once a prewrite-lock is converted to a write-lock, no other operation can lock any data object.*
2. *for any operation* $pul_i \in T_i$, $(pwl_i(x) \to wl_i(x)) < pul_i(x)$. *That is, a prewrite-lock is converted to a write-lock before any lock is released.*

Property 4. *If* $p_i(x)$ *and* $q_j(x)$ *are two conflicting operations then either*

1. $pul_i(x) <_H ql_j(x)$. *If* $p_i(x)$ *is a prewrite operation then* $(pwl_i(x) \to wl_i(x)) <_H ql_j(x)$. *If* $p_i(x)$ *is read operation then* $pul_i(x) <_H (pwl_j(x) \to wl_j(x))$. *or*
2. $qul_j(x) <_H pl_i(x)$. *If* $q_j(x)$ *is a prewrite operation then* $(pwl_j(x) \to wl_j(x)) <_H pl_i(x)$. *If* $q_j(x)$ *is a read then* $qul_j(x) <_H (pwl_i(x) \to wl_i(x))$.

Property 5. *If* pc_i *and* c_i *are pre-commit and commit then* $pc_i <_i c_i$ *in* T_i *and for any* pc_j, *if* $pc_i < pc_j$ *then* $c_i < c_j$. *However, if the transaction* T_j *is a read-only transaction and* $pc_i < pr_j$ *then* $c_i < c_j$ *or* $c_j < c_i$.

Property 6. *If* $pc_i \in T_i$ *then no* $a_i \in T_i$.

Property 7. *If* $pw_i(x), w_i(x) \in T_i$ *and* $pw_j(x), w_j(x) \in T_j$ *and if* $T_i \to T_j$ *then*

1. *if* $pwl_i(x) < pwl_j(x)$ *then* $wl_i(x) < wl_j(x)$.
2. *If* $wl_i(x) < pwl_j(x)$ *then* $pwl_j(x) < w_i(x) < wl_j(x)$ *or* $wl_i(x) < pw_j(x) < w_j(x)$.

We now prove that any history that satisfies the above properties has an acyclic serialization graph. for a history H, SG(H) has a node for each transaction and an edge $T_i \to T_j$ if T_i has an operation p_i that conflicts with an operation $q_j \in T_j$.

Lemma 1. *If* $T_1 \to T_2$ *in SG(H) then there exists an unlock operation* $pul_1 \in T_1$ *or a lock convert operation* $(pwl_1 \to wl_1) \in T_1$ *such that for all lock operations* $ql_2 \in T_2$ *or a lock convert operation* $(pwl_2 \to wl_2) \in T_2$, $pul_1(x) < ql_2(x)$ *or* $(pwl_1(x) \to wl_1(x)) < ql_2(x)$ *or* $pul_1(x) < (pwl_2(x) \to wl_2(x))$.

Proof: Since $T_1 \to T_2$, there must exist conflicting operations $p_1(x)$ and $q_2(x)$ such that $p_1(x) < q_2(x)$. By Property 1, we have the following

(a) If $p_1(x)$ is a read or pre-read operation then $pl_1(x) < p_1(x) < pul_1(x)$. If $p_1(x)$ is a prewrite operation then $(pwl_1(x) < p_1(x) < (pwl_1(x) \to wl_1(x))$. Otherwise, if p is a write operation then $(pwl_1(x) \to wl_1(x)) < p_1(x) < pul_1(x)$.

(b) If $q_2(x)$ is a read or pre-read operation then $ql_2(x) < q_2(x) < qul_2(x)$. If $q_2(x)$ is a prewrite operation then $(pwl_2(x) < q_2(x) < (pwl_2(x) \rightarrow wl_2(x))$. Otherwise, if $q_2(x)$ is a write operation then $(pwl_2(x) \rightarrow wl_2(x)) < q_2(x) < qul_2(x)$.

By property 4, we have the following

1. $pul_1(x) < ql_2(x)$ or $(pwl_1(x) \rightarrow wl_1(x)) < ql_2(x)$ or $pul_1(x) < (pwl_2(x) \rightarrow wl_2(x))$.
2. $qul_2(x) < pl_1(x)$ or $(pwl_2(x) \rightarrow wl_2(x)) < pl_1(x)$ or $qul_2(x) < (pwl_1(x) \rightarrow wl_1(x))$.

From (2), and using (a) and (b), we get $q_2(x) < qul_2(x) < pl_1(x) < p_1(x)$ which contradicts that $p_1(x) < q_2(x)$.

Again, if q_2 is a prewrite operation then we get, either $q_2(x) < (pwl_2(x) \rightarrow wl_2(x)) < pl_1(x) < p_1(x)$ which again contradicts that $p_1(x) < q_2(x)$ or If q_2 is a write operation then we get $(pwl_2(x) \rightarrow wl_2(x)) < q_2(x) < qul_2(x) < pl_1(x) < p_1(x)$ which again contradicts that $p_1(x) < q_2(x)$. □

Lemma 2. *Let $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$ be a path in SG(H) where $n > 1$. Then for data objects x and y and some operations $p_1(x)$ and $q_n(y)$ in H, $pu_1(x) < ql_n(y)$ or $(pwl_1(x) \rightarrow wl_1(x)) < ql_n(y)$ or $pul_1(x) < (pwl)_n(y) \rightarrow wl_n(y))$.*

Proof: By mathematical induction, for $n = 2$, it follows from Lemma 1. Assume that the lemma holds for some $k \geq 2$. we will prove that it holds for $n = k + 1$. By the induction hypothesis, the path $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_k$ implies that there exist data items x and z and operations $p_1(x)$ and $o_k(z)$ in H such that $pul_1(x) < ol_k(z)$ or $(pwl_1(x) \rightarrow wl_1(x)) < ol_k(z)$ or $pul_1(x) < (pwl_k(z) \rightarrow wl_k(z))$. By $T_k \rightarrow T_{k+1}$, and by Lemma 1, there exists a data item y and conflicting operations $o'_k(y)$ and $q_{k+1}(y)$ in H such that $o'ul_k(y) < ql_{k+1}(y)$ or $(pwl_k(y) \rightarrow wl_k(y)) < ql_{k+1}(y)$ or $pul_1(y) < (pwl_k(y) \rightarrow wl_k(y))$. By Property 2, we have $ol_k(z) < o'ul_k(y)$. By Property 3, we have $ol_k(z) < (pwl_k(y) \rightarrow wl_k(y))$ or $(pwl_k(z) \rightarrow wl_k(z)) < pul_k(y)$. On combining the above, we get

1. $pul_1(x) < ol_k(z) < o'ul_k(y) < ql_{k+1}(y)$. This implies that $pul_1(x) < ql_{k+1}(y)$ as desired.
2. $(pwl_1(x) \rightarrow wl_1(x)) < ol)_k(z) < (pwl_k(y) \rightarrow wl_k(y) < ql_{k+1}(y)$. This implies that $(pwl_1(x) \rightarrow wl_1(x)) < ql_{k+1}(y)$ as desired.
3. $pul_1(x) < (pwl_k(z) \rightarrow wl_k(z)) < pul_k(y) < (pwl_k(y) \rightarrow wl_k(y)) < ql_{k+1}(y)$. This implies that $pul_1(x) < ql_{k+1}(y)$. □

Theorem 1. *Every history H obtained by the locking protocols given before is serializable.*

Proof: Suppose by the way of contradiction that SG(H) has a cycle $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$ where $n > 1$. By Lemma 2, for some data objects x and y and some operations $p_1(x)$ and $q_1(y)$ in H, $pul_1(x) < ql_1(y)$ or $(pwl_1(x) \rightarrow wl_1(x)) < ql_1(y)$. But this contradicts Property 2. Thus, SG(H) has no cycle and so by the serializability theorem [25], H is serializable. □

7. Discussion

In this section, first we discuss overview of our recovery model. Next, we briefly present some intuitive analysis of our model and mainly investigate the parameters, which can affect the mobile transaction processing with respect to our model (we have simulated some of these parameters in Section 8 where we report the performance results of our model). Next, we briefly give some details of implementation environment of our model.

7.1. Recovery model

When executing transactions in mobile computing, it is necessary to maintain logs to enable recovery after a system crash. A MH is highly vulnerable to failures due to the loss or theft of equipment, memory loss, etc. In order to recover from such failures, it is necessary to store data objects and their logs at the mobile service stations (MSS) rather than on mobile host. A MH in our model transfers a transaction's execution to the MSS by moving all the prewrite values and the log records. A separate pre-commit log is maintained for each transaction. Each log record contains the description of the prewrite values. These are appended to the pre-commit record of a transaction. Thus, for every pre-committed transaction, the pre-commit log records for that transaction are stored on the disk as well as in the system log at MSS.

In case a MH moves to a different cell, it can also append a record indicating its next possible destination. When the user arrives at the new cell, he can continue the post pre-commit operations. Once a transaction's execution is successfully shifted (along with all prewrite logs and pre-commit log) to the MSS, a MH may delete all the log records and keep only prewrite values in main memory for further processing. This way it can gradually discard entries in prewrite logs. To build highly reliable systems, these logs can also be replicated in various cells (MSS) by moving MH. At MSS, prewrite-logs, pre-commit log record, the write logs and final commit log record are stored. If a pre-committed transaction fails at the stationary host, the transaction can be restarted from the point of failure. In case of a system crash, the prewrite logs can be used to build the system's state as it existed at the time of pre-commit. Write logs can help to bring the system to the state as it existed at the time of failure with the help of recovery algorithms ([16, 21]). Since the locks are managed by MSS, the locks are maintained at MSS. The transaction table for active transactions are kept at MSS. However, in case the transaction starts its execution at MH then a part of the transaction table is also maintained at MH until the transaction's execution is shifted to the MSS. Dirty-data object table is kept at MSS. This table contains information about those objects whose final values are inconsistent with the stable database on the disk. This also keeps information about those data objects whose prewrite values announced by the pre-committed transactions have not been updated in the database before system crash. In Table 2, we give the type of log records and the tables stored at MH and MSS. Some of these logs are moved to MSS from MH during the shift of a transaction's execution.

In case of a system crash, only the redo of those prewrite and write values will be performed whose effects are not there on stable storage. There is no need for undo pass of the recovery algorithm as data objects are updated only after the transaction is pre-committed.

Table 2. Log records at MH and MSS.

Log records stored at MH	Log records stored at MSS
Prewrite logs	Write log
Pre-commit log	Commit log
'Destination' move log	Lock. dirty object and transaction tables

For more details of no undo/redo recovery algorithm, refer to [21] where a nested transaction recover model is described.

7.2. Evaluation of prewrite operations

How prewrite mobile transaction model help in dealing with constrained resources and frequent disconnections or weak connections. In mobile computing, to save the energy and to deal with unpredictable failure of mobile host, transactions should be migrated to non-mobile computer in case no further interaction is needed with MH. In case the mobile host transfers the remaining transaction's execution after pre-commit to the stationary host (MSS), it submits all the prewrite values and write operations to the stationary host. The transaction at the stationary host updates prewrite data objects and commits. For example, consider a broker who is travelling. While travelling, he would like to place an order for stock purchase. He initiates a buy transaction by submitting the price and the quantity of the stock. He places the order without checking his account balance, as he has to pay only after three working days of purchase. He also knows that the payment of his earlier sold stock will arrive during this period. This arrival of "payment" represents a final commit of his earlier transaction. Thus, selling and buying transactions are running concurrently. Since the transaction is very long, placing the order helps in pre-committing the transaction at the MH and remaining execution (buying of stock and payment etc.) is shifted to the MSS. This works well even if the MH is disconnected or it is in doze state.

Do prewrites help in caching data during weak-connections? A mobile host can cache only the prewrite values of the data objects for later processing. For example, a "bed-reservation" transaction at mobile host in Section 3 needs only the "headline-report" to found about the number of people injured. It does not need the full story. Moreover, the consistency of prewrite and write values can also be tolerated depending on applications. Thus, it is not required that cached prewrite values at MH and write values at MSS need to be consistent all the time. If the ratio (size of prewrite-value/size of write-value) becomes smaller then it can be transmitted over low band-width with less power consumption and low-cost. Headline-report, for example, will use less memory and battery power and therefore, can be transmitted over low bandwidth. Similarly, a model of an "engineering design" can be transmitted faster over low bandwidth rather than the complete design. Consider that if the size of the complete design is 10 Mb and the size of model design is 2 Mb and the bandwidth is 1 Mbps (Infrared, see Table 3). In this case, the network delay on average in case of transmission of complete design from MH to the MSS will be 10 seconds

Table 3. Simulation parameters.

System parameters	Description	Value
DB-size	Average size of the database	500
Num-MH	Number of MHs	Simulation parameter
Num-MSS	Number of MSSs	2
Trans-size	Average number of objects per transaction	6 objects
Pre-value-size	Average size of pre-write values	1/40 of write value
Max-size	Maximum number of objects per transaction	10 objects
Min-size	Minimum number of objects per transaction	2 objects
Local-object-MH	Ratio of objects found in cache at MH	0.4
CPU-time	Time taken by CPU per request	12 msec
I/O time	Time taken by I/O per request both at MSS and MH	30 msec
Num-transactions-MSS	Transactions at each MSS	Simulation parameter
Wireless-bandwidth	Data transfer rate from MH to MSS	0.5 Mbps
Write-prob	Probability that object read will be written also	Simulation parameter
Trans-delay	Inter-arrival delay time	500 msecs
Prewrite-to-write	Delay in write	0.2 msecs

whereas the delay will be 2 seconds in case of transmission of model design. This will decrease the communication cost. Prewrites can be cached during weak connections over low bandwidth due to their small size. This will minimise future network use by MH and improve throughput. Furthermore, sending access requests from the mobile host to the MSS may be expensive due to the limited up link bandwidth and it also uses considerable amount of portable battery power. Thus, by maintaining prewrites at MH can alleviate this problem.

If the user is to be charged based on per time unit of connection time, the sending or receiving prewrite values will incur less cost than the write values. Similarly, if the user is charged per message basis depending on the length of the message, it will cost less to send or receive prewrite model of the design rather than the final design. Since transmitting data consumes more battery power [12], the sending of the model design will be more cost-effective than sending the completed design. Next, if the read/write ratio becomes greater then prewrite will be more useful in mobile computing environment.

Do prewrites avoid undo cost in case of transaction aborts? Our model does not need to handle aborts by using before-image or compensation. A pre-committed transaction does not abort. In case a pre-committed transaction is forced to abort due to system dependent reasons such as crash, the transaction will be restarted. For example, an accident-report transaction discussed in Section 4 need not abort in the case of a failure. In various proposed mobile transaction models, an abort is handled by a compensating transaction. This is costly in mobile computing, as it needs extra resources and thus, increases computing cost. In our model, since a transaction's execution is shifted to the MSS after pre-commit, the restart cost will be born by the MSS.

Do prewrites help in continuing the execution without blocking the transactions and data objects in case of frequent disconnection? In case the mobile host is disconnected or moved to a different cell, further processing at MH can still continue using prewrites. It need not wait for the commit of earlier transaction at the stationary host. At the same time, using prewrite values, other transactions at stationary host can continue their execution in case of a disconnection from MH. Since the prewrites are available at MH, it does not require to cache the values again. Thus, in our model, the partial execution of transaction at the mobile host and the completion of the transaction after pre-commit at the stationary host increases availability with reduced execution cost.

Does delay in writing the database helps in reducing network congestion? In our mobile transaction model, final updates to be performed at MSS are delayed. This action has some advantages in mobile computing. Since write operations are time and resource consuming, they are transferred to MSS for updates on the database. Delayed writes reduce network traffic congestion generally caused by frequent online updates and can speed up the read requests. The weak-connections in a network can force a mobile host to wait for long in case it would like to update database at MSS. Delayed writes also reduce the work-load on servers since the transactions updates are synchronised at the time of pre-commit.

Do prewrites simplify the work of hand-off protocol? Prewrites reduce the work of hand-off protocol as a pre-committed transaction's execution does not require migration to the new MSS in case MH moves to the new cell. The transaction can still continue at the old MSS due to the fact that the transaction needs no more interactions with MH. If a transaction has not pre-committed and moved to the new cell, it can still continue its processing in the new cell and can pre-commit there.

7.3. Implementation overview

Mobile computing must operate in an environment where network connectivity, power and contextual information should vary. Currently we are building our mobile transaction processing system at RAID lab at Purdue University. Raid lab has 5 SUN 3/50s and four Sun Sparcstations-1s. All of them have local disks and are connected by a 10 Mb/s Ethernet. First our goal is to build our transaction processing system within existing client-server distributed system called *Raid*. Next, our goal is to build a software emulator over the *Raid* [5]. The emulator is designed to provide approximate emulation of wireless network using the standard hardware and software. The basic approach is to intercept the packets travelling between client and server and to introduce a delay, provide a disconnection and variable bandwidth etc. The network emulator supports dynamic updates in the configuration file so that quality of service of a network can be set to zero to emulate disconnection.

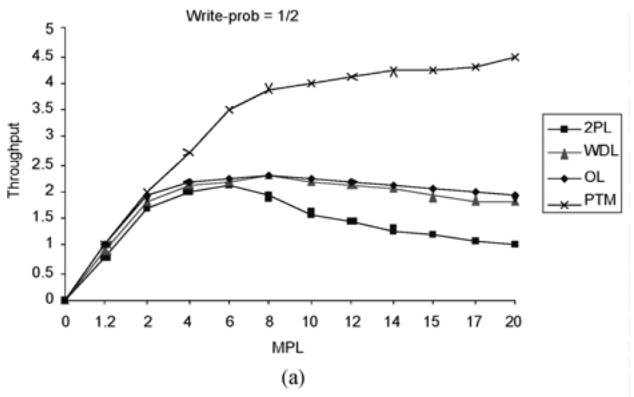
Raid is a robust and adaptable distributed system for transaction processing. The client server architecture of Raid provides modularity and extensibility, which we need to support mobile computing environment. Raid system interacts with underlying data via read/write requests and init-transaction and commit-transaction. We have modified the types of operations by including pre-read and prewrite operations and also included a pre-commit semantics to support our transaction model. A front end invoked by the user at the client side (one of the terminal connected to Raid depicts as Mobile Host) to process SQL++ queries. The SQL++ query processor parses the SQL++ language query received from the user into a sequence of pre-read, read, prewrite and write relation tuple requests submits these requests to the Raid server (one of the terminal which now depicts as Mobile Service Station). We assume that a copy of the data object (a relational table) is there at MH. Query processor generates a transaction consisting of operations on tuples of the table and submits them to transaction manager at mobile host. TM at the mobile host supervises the states of mobile transactions going on and communicates with the mobile transaction manager agent at MSS. At the MSS side, a mobile transaction manger agent services a particular mobile host by receiving all the operations. It acts for transaction management that a mobile transaction manager has requested. It returns to the mobile transaction manger at MH the requested lock for pre-write operation. On receiving the lock, the TM at MH announces the prewrite values of the tuples of the table at the private workspace in TM and also writes the log records at its local disk and also send them to Raid server (MSS). The mobile service station (Raid server) consists of four parts, the Replication Controller, the Concurrency Controller, Atomicity Controller and Data Access Manager. The replication controller manages multiple copies of the data items to provide system reliability and mutual consistency of the data. The concurrency controller ensures serializability. The atomicity controller is responsible for ensuring that transactiona are committed or aborted across MH and MSS. The data access manager provides access to the database at MSS (Raid server) and ensures that updates are posted atomically to stable storage. Once the mobile transaction manager agent informs the MH commit of the transaction, MH cleans its log and updates the table. In case the table is not present at the MH, the raid server simply send those tuples which involves in the prewrite operations along with the lock.

A layered communication package in RAID provides a clean location independent interface between the servers. However, in mobile computing environment the queries are location dependent and location data plays an important role. In the emulator, we currently looking into the methods of emulating locations change. Also, we would like to incorporate in the emulator the hand-off protocol.

8. Performance

In this section, we develop a simulation model and compare our model with that of two phase-locking algorithm [4] and Optimistic locking protocols [1, 32] and wait-depth-limited scheme [10]. Since our model is lock-based, we consider only lock-based concurrency control algorithms. We assume close system for our simulation where we always keep certain fixed number of active transactions in the system. The simulation model has the following components: mobile host transaction manager (MHTM), mobile service station transaction Manager (MSSTM), mobile host data manager (MHDM), mobile service station data manager (MSSDM), mobile service station lock manager (MSSLM). MHTM generates and manages the transactions initiated at MH where as similar role is played by its counter part MSSTM at server side. We assume that messages send by sender have always been received by receiver. Both MHTM and MSSTM generate transactions which controls the multi-programming level in the system. We assume multiple MHs in the cell and 2 MSS. Multiple CPU servers are used to perform transaction executions both at MH and MSS. When a transaction needs to be executed, a free server is assigned from global queue in FCFS discipline. Each MH and MSS also has I/O servers and there is a queue associated with I/O servers. Each DM also has a number of disk servers. When a transactions needs service, it can randomly choose a disk a waits in I/O queue. The service is again FCFS. In case of transaction execution at MH, the I/O wait time is negligible as only one transaction at a time is executed. The CPU and disks are allocated at each MH and MSS. Each consists of one CPU, and each MH consists of 1 disk where as MSS consists of 2 disks. Major simulation parameters are shown in Table 3. The primary performance metrics is throughput rate; the number of transactions that commit per second. Second parameter we study is transaction-abort-ratio; the average number of transactions that have to be aborted per total transactions. The lower transaction-abort-ratio means enhanced concurrency.

The meaning of each model parameter for simulation is as follows. The size of the database assumed is DB-size data objects. We assume that each data object is equivalent to a page. Num-MH is the number of mobile hosts, which effectively controls the multi-programming level (MPL). Thus, it is a simulation parameter. Num-MSS is the number of mobile service stations. We assume there are two MSS in the system. Therefore, the total number of sites in the system is num-MH+2. Num-transactions-MSS is the number of transactions running at a time in MSS. It is a simulation parameter. Thus, the total number of transactions in the system at a time is num-MH + Num-transactions-MSS since each MH executes one transaction at a time. Trans-size is the average number of data objects requested by the transaction. Pre-value-size is the size of prewrite-data object in comparison to the write-data object. The max-size and min-size are the maximum and minimum number of objects per transaction. Write-prob is the probability that an object read by a transaction will also be written by the same transaction. It is also a simulation parameter. Local-object-MH is the ratio of objects found in cache at MH. That is, if a transaction writes m data objects and n ($<$m) objects are available at MH then local-object-MH is n/m. CPU-time and I/O time are the time taken to carry out CPU and I/O requests respectively. These are taken into consideration when an object is accessed. To force-write a log record requires CPU-time and I/O time. Also, to write each data object to the disk requires both CPU-time and I/O-time. Trans-delay is transmission delay from MH to MSS. The local-object-MH is fixed for each transaction. Whenever, a lock requests wait, deadlock is detected. However, we neglect the deadlock overheads since they are negligible as compared to overall cost. Aborted transaction is resubmitted after a delay and makes same data access. The length of delay is equal to the average response time, which is used in the most transaction processing studies.



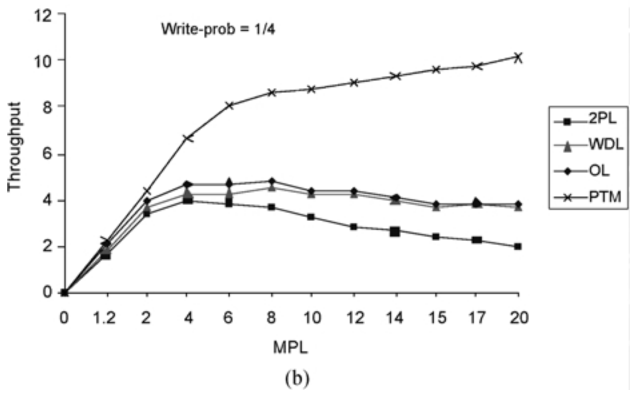


Figure 8. (a). Throughput v/s MPL (b). Throughput v/s MPL.

8.1. Interpretation of results

In our experiments, we investigate the performance of our concurrency control algorithm with two phase locking (2PL), wait-depth locking (WDL) and optimistic locking algorithm (OL). In the first experiment (see figure 8(a)), the write-prob is set to $\frac{1}{2}$. That is, half of the total transactions submitted are always read and write data items. Also, in this experiment we assume that transaction arrives at MH and is executed only at MSS. In this case, we observe that throughput increases rapidly for our prewrite transaction model with increased MPL in comparison with other schemes. The reason is that prewrite-locks are released earlier, so read-only transactions do not wait for the write transactions to release the locks. Also, transactions commit can be decided at pre-commit time. Moreover, read actions can read the value written before the commit of transactions as write-locks are released before final commit. In second experiment (see figure 8(b)), we lower the write-prob to 1/4. That is, only 25% of total transactions perform both read and write operations. In other words, the read-only transactions are increased by 50% than last experiment. In this case we assume that transactions are also executed at MHs. We observe that now throughput increases more

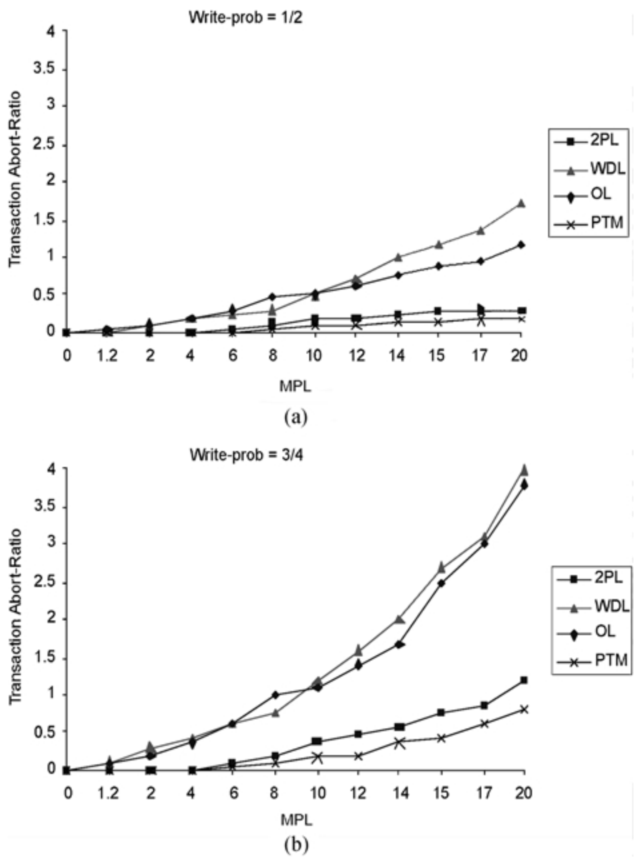


Figure 9. (a). Transaction-Abort-ratio v/s MPL (b). Transaction-Abort-ratio v/s MPL.

rapidly as compared to other algorithms. The reason being, the read-only transactions can read prewrite values at MH and can commit. Thus, our algorithm performs much better in terms of read-only transactions. Also, prewrite values take less time to travel from MSS to MH than write values as in case of other schemes.

In the third experiment (figure 9(a) and (b)), we have observed that transaction-abort-ratio (total aborts divided by the number of submitted transactions) decreases more rapidly for our model than others. The reason is that after pre-commit, transactions in our model do not abort and therefore, those transactions who read prewrite values, are guaranteed to commit. Similarly, write transactions are ordered with respect to pre-commit order, thus, they do not involve in deadlocks and hence, aborts are avoided. Thus, deadlocks in our model are only possible before transactions hold write-locks. We have done experiments by keeping the level of write-prob as $\frac{1}{4}$ and $\frac{3}{4}$. The transaction-abort-ratio increases with increase in write-prob. That is, the transaction-abort-ratio with more number of write transactions rises, as expected. This is more in case of WDL and OL protocols due to more number of conflicts in case of write operations and thus, more abortions.

Thus, our protocol outperforms all the three other protocols in terms of throughput and transaction-abort-ratio.

9. Conclusions

In this paper, we have presented a new mobile transaction model using prewrites to increase availability in mobile computing environment. Prewrite values help in increasing availability as the transactions can be executed during disconnections both at MH and MSS without blocking. The model allows a transaction's execution to shift from the MH to MSS for database updates. Thus, reduces the computing expenses. The model needs no "undo" actions or execution of compensating transaction in case of transaction aborts. We have discussed algorithms for our transaction processing model and the locking protocols. We have proved that our mobile transaction model produces only serializable schedules. We performed simulation studies and have found that our model increases throughput in comparison with other models. Also, our model has higher commit probability; that is low transaction-abort ratio, thus further enhance concurrency. For future work, we would like to discuss a detail crash recovery algorithm for mobile transaction model.

Currently, we are implementing our transaction model at RAID lab, Department of Computer Science at Purdue University, West Lafayette. We are creating a wireless network emulator by dynamically breaking the link, providing variable bandwidth, etc. to emulate the wireless environment. We plan a series of real experiments with our model and to compare it with other existing transaction models for mobile computing.

Acknowledgments

This research is partly supported by a grant from NSF under 9805693-EIA and CCR-9901712.

References

1. D. Agrawal, A.El. Abbadi, and A.E. Lang, "The performance of protocols based on locks with ordered sharing," IEEE Transactions on Knowledge and Data Engineering, vol. 6, no. 5, pp. 805–818, 1994.
2. D. Agrawal and S. Sengupta, "Modular synchronization in multiversion databases: Version control and concurrency control," in ACM Proceedings of SIGMOD, ACM Press: New York, 1989, pp. 408–417.
3. N. Barghouti and G. Kaiser, "Concurrency control in advanced database applications," ACM Computing Surveys, vol. 23, no. 3, pp. 269–317, 1991.
4. P. Bernstein, V. Hadzilacos, and N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley Publishing Co., USA, 1987.
5. B. Bhargava and J. Riedl, "A model for adaptable systems for transaction processing," IEEE Transaction on Knowledge and Data Engineering, vol. 1, no. 4, pp. 433–449, 1989.
6. P. Bober and C.J. Michael, "On mixing queries and transactions via multiversion locking," Computer Science Department, University of Wisconsin-Madison, Technical Report, Nov., 1991.
7. A. Chan, S. Fox, W. Lin, A. Nori, and D. Ries, "The implementation of an integrated concurrency control and recovery scheme," in ACM Proceedings of SIGMOD, ACM Press, New York, 1982, pp. 184–191.
8. P.K. Chrysanthis, "Transaction processing in a mobile computing environment," in Proceedings of IEEE Workshop on Advances in Parallel and Distributed Systems, 1993, pp. 77–82.

9. M.H. Eich and A. Helal, "A mobile transaction model that captures both data and movement behaviour," ACM/Baltzer Journal on Special Topics on Mobile Networks and Applications, vol. 2, no. 2, pp. 149–162, 1997.
10. P.A. Franaszek, J.T. Robinson, and A. Thomasian, "Concurrency control for high contention environments," ACM Transactions on Database Systems, vol. 17, no. 2, pp. 304–345, 1992.
11. S. Goel, B. Bhargava, and S.K. Madria, "An adaptable constrained locking protocol for high data contention environments," in Proceedings of IEEE for 6th Intl. Conference on Database Systems for Advanced Applications (DASFAA,99), Taiwan, 1999.
12. T. Imielinksi and B.R. Badrinath, "Wireless mobile computing: Challenges in data management," Communications of ACM, vol. 37, no. 10, pp. 18–28, 1994.
13. R. Kataoka, T. Satoh, and U. Inoue, "A multiversion concurrency control algorithm for concurrent execution of partial update and bulk retrieval transactions," in Proceedings 10th Intl. Phoenix Conference on Computers and Communications, IEEE Computer Society Press: New Jersey, 1991, pp. 130–136.
14. J. Kistler and M. Satyanarayanan, "Disconnected operation in the coda file system," ACM Transactions on Computer Systems, vol. 10, no. 1, pp. 3–25, 1992.
15. Q. Lu and M. Satyanaraynan, "Improving data consistency in mobile computing using isolation-only transactions," in Proceedings of the Fifth Workshop on Hot Topics in Operating Systems, Washington, 1995.
16. S.K. Madria, "Concurrency control and recovery algorithms in nested transaction environment and their proofs of correctness," Ph.D. Thesis, Department of Mathematics, Indian Institute of Technology, Delhi, 1995.
17. S.K. Madria, "A prewrite transaction model," in the Proceedings of 3rd International Baltic Workshop on Database and Information Systems, Riga, Latvia, 1998.
18. S.K. Madria, "Transaction models for mobile computing," in Proceedings of 6th IEEE Singapore International Conference on Network, World Scientific, Singapore, 1998.
19. S.K. Madria and B. Bhargava, "System defined prewrites to increase concurrency in databases," in Proceedings of the First East-Europian Symposium on Advances in Databases and Information Systems, ADBIS'97 (sponsored by ACM-SIGMOD), St.-Petersburg (Russia), Sept. 97, pp. 18–22.
20. S.K. Madria and B. Bhargava, "A transaction model for mobile computing," in IEEE CS Proceedings of International Database Engineering and Application Symposium (IDEAS'98), Cardiff, U.K., 1998.
21. S.K. Madria, S.N. Maheshwari, B. Chandra, and B. Bhargava, "Crash recovery algorithm in an open and safe nested transaction model," in Proceedings of 8th International Conference on Database and Expert System Applications (DEXA'97), France, Sept.97, Lecture Notes in Computer Science, vol. 1308, Springer Verlag, France.
22. S.K. Madria and M. Mohania, "A study on mobile data and transaction management," Research Report CIS-98-007, Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, Adelaide, Australia, June, 1998.
23. J.E.B. Moss, "Nested transactions: An approach to reliable distributed computing," Ph.D. Thesis, also, Technical Report MIT/LCS/TR-260 MIT Laboratory for Computer Science, Cambridge, MA., April, 1981.
24. T. Nakajima, "Commutativity based concurrency control for multiversion objects," in Proceedings of the International Workshop on Distributed Object Management, 1992, pp. 101–119.
25. C.H. Papadimitriou, "The serializability of concurrent database updates," Journal of ACM, vol. 26, no. 4, pp. 631–653, 1979.
26. E. Pitoura and B. Bhargava, "Dealing with mobility: Issues and research challenges," Technical Report TR-93-070, Department of Computer Sciences, Purdue University, IN, 1993.
27. E. Pitoura and B. Bhargava, "Building information systems for mobile environments," in Proceedings of 3rd International Conference on Information and Knowledge Management, 1994, pp. 371–378.
28. E. Pitoura and B. Bhargava, "Maintaining consistency of data in mobile computing environments," in Proceedings of 15th International Conference on Distributed Computing Systems, June, 1995. Extended version to appear in IEEE TKDE, 2000.
29. C. Pu, G. Kaiser, and Hutchinson, "Split-transactions for open-ended activities," in Proceedings of the 14th VLDB Conference, 1988.
30. C. Pu and A. Leff, "Replica control in distributed systems: An asynchronous approach," in Proceedings of the ACM SIGMOD, 1991, pp. 377–386.

31. K. Ramamritham and P.K. Chrysanthis, “A taxonomy of correctness criterion in database applications,” Journal of Very Large Databases, vol. 5, no. 1, pp. 85–97, 1996.
32. K. Salem, H. GarciaMolina, and J. Shands, “Altruistic locking,” ACM Transactions on Database Systems, vol. 19, no. 1, pp. 117–165, 1994.
33. G.D. Walborn and P.K. Chrysanthis, “Supporting semantics-based transaction processing in mobile database applications,” in Proceedings of 14th IEEE Symposium on Reliable Distributed Systems, 1995, pp. 31–40.
34. W.E. Weihl, “Distributed version management for read-only actions,” IEEE Transactions Software Engineering, vol. 13, no. 1, pp. 55–64, 1987.
35. W.E. Weihl, “Commutativity-based concurrency control for abstract data types,” IEEE Transactions on Computers, vol. 37, no. 12, pp. 1488–1505, 1988.
36. S.K. Madria, M. Mohania, S. Bhowmick, and B. Bhargava, “A survey on mobile data and transaction management issues,” under revision in Information Science Journal.