A transaction model and multiversion concurrency control for mobile database systems

Sanjay Kumar Madria · M. Baseer · Vijay Kumar · Sourav Bhowmick

Published online: 30 October 2007
© Springer Science+Business Media, LLC 2007

Abstract Transaction management on Mobile Database Systems (MDS) has to cope with a number of constraints such as limited bandwidth, low processing power, unreliable communication, and mobility etc. As a result of these constraints, traditional concurrency control mechanisms are unable to manage transactional activities to maintain availability. Innovative transaction execution schemes and concurrency control mechanisms are therefore required to exploit the full potential of MDS. In this paper, we report our investigation on a multi-versions transaction processing approach and a deadlock-free concurrency control mechanism based on *multiversion two-phase* locking scheme integrated with a timestamp approach. We study the behavior of the proposed model with a simulation study in a MDS environment. We have compared our schemes using a reference model to argue that such a performance comparison helps to show the superiority of our model over others. Experimental results demonstrate that our model provide significantly higher throughput by improving degree of concurrency, by reducing transaction wait time, and by minimizing restarts and aborts.

Keywords Mobile transaction · Concurrency · Multiversions · Locking · Timestamps

Communicated by Ahmed K. Elmagarmid.

S.K. Madria (✉) · M. Baseer
Department of Computer Science, University of Missouri-Rolla, Rolla, MO 65401, USA
e-mail: madrias@umr.edu

V. Kumar
SICE, Computer Networking, University of Missouri-Kansas, Kansas City, MO 64110, USA
e-mail: kumarv@umkc.edu

S. Bhowmick
School of Computer Engineering, Nanyang Technological University, Singapore, Singapore

1 Introduction

A Mobile Database System (MDS) allows its clients to initiate and process transactions from anywhere and at anytime. MDS, which we envision, has a number of applications and system level problems, which must be solved before MDS can be fully realized. One of such complex problems is maintaining database consistency in the presence of high contention transaction traffic. Maintaining consistency has always been a resource intensive process, however, it gets worse in MDS because of a number of factors related to its architecture, availability and sharing of hardware and software resources, distribution of data, and mobile client's processing capability. One of the serious concerns is the efficient utilization of limited wireless channels to provide acceptable level of performance [4].

Some of the problems in managing transaction and data in a mobile environment have been identified in (we list only a few here) [5, 11, 13, 35, 42]. We revisit these limitations and justify the need for developing an efficient transaction processing scheme, augmented with a concurrency control and a serialization technique for Mobile Transactions (MT) running on MDS. We first identify some unique situations which MT's may encounter during their execution, and which do not appear in conventional database systems [30, 42].

- A short MT (accessing only a few data items) could become long-lived due to the mobility of both the data and users, and due to frequent disconnections. Note that long-lived here does not mean MT accesses a large number of database items, rather it takes longer to finish the execution.
- Transactions in MDS are executed in a distributed manner (i.e., client-server) which may be subjected to further restrictions due to inherent constraints such as limited bandwidth. Distributed execution of transactions in the presence of mobility makes transaction commitment and termination quite complex [20].
- A MT may have to suffer "forced wait" or "forced abort" because of the lack of wireless channels (uplink or downlink) and it may be delayed further due to random hand-offs. In addition, a MT at mobile host may not be able to complete its execution due to non-availability of complete DBMS capability.

In this paper, we present a transaction processing model augmented with a multiversion concurrency control mechanism (CCM) to increase resource availability and reduced aborts in MDS. Our CCM aims to achieve high throughput and low-abort rate by increasing data availability during concurrent read-write and write-write operations, and by reducing the restart rate. Note that reducing aborts is very important in mobile networks [8]. A short-version of this paper appeared in [28] which did not include formal model, experimental evaluation and proof of correctness. We compare the performance of our protocol with a speculative model for transaction processing (SMTP) using model that appeared in [38, 39]. In this reference model, each transaction is executed based on both after and before images and each transaction commits based on the outcome of the previous transaction. Note that this is the closet model we found which uses versions in a mobile environment and has some similarity with our scheme. Moreover, this model outperforms basic 2PL [15], and WDL [7] in a similar environment. We justify based on the experimental results that our model presented in this paper outperforms the SMTP model in terms of response time and throughput.

2 Architecture of mobile database system (MDS)

The architecture of the Mobile Database System (MDS) we are investigating is shown in Fig. 1. It is a distributed client/server system based on the Personal Communication Systems (PCS). To incorporate distributed database functionality, we have added a number of DBSs (database Servers) without affecting any aspect of the generic mobile network [23]. The entire platform is comparable to a distributed multidatabase system with a special feature of mobility in data processing.

The platform is composed of a set of general purpose computers (PCs, workstations, etc.) which are interconnected through a high-speed wired network. These computers are categorized as Fixed Hosts (FH) and Base Stations (BS) or Mobile Support Stations (DBS). A number of mobile computers (laptop, PDAs, etc.) referred to as Mobile Hosts (MH) or Mobile Units (MU) are connected to the wired network components only through BSs via wireless channels. A BS maintains and communicates with its MUs and has some processing capability. One or more BSs are connected with a BSC (Base Station Controller or Cell Site Controller), which coordinates the operation of its BSs using its own stored software program when commanded by the MSC (Mobile Switching Center). The MSC is connected to the PSTN (Public Switched Telephone Network). MUs are battery powered portable computers, which move around freely in a restricted area, which we refer to as the "geographical mobility domain" (G). For example in Fig. 1, G is the total area covered by all BSs. This size restriction on their mobility is mainly due to the limited bandwidth of wireless communication channels.

To support the mobility of MHs and to exploit frequency reuse, the entire G is divided into smaller areas called cells. Each cell is managed by a particular BS and is allocated a set of frequencies for communication. The mobility support requires that an MH must have unrestricted movement within G (inter-cell movement) and must be able to access desired data from any cell. The process of crossing a cell boundary

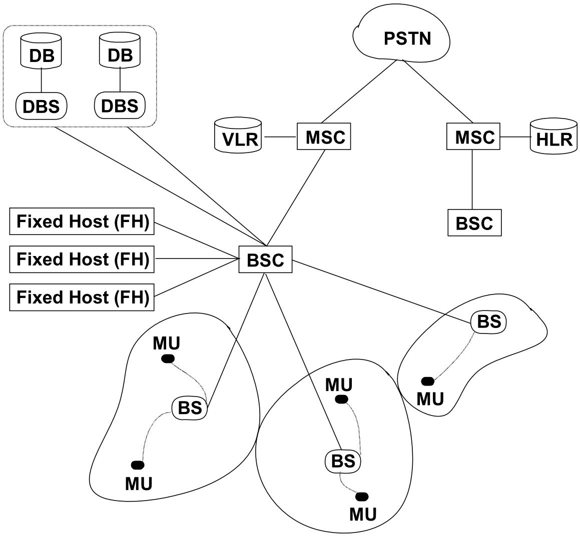


Fig. 1 Reference architecture of mobile database system (MDS)

by an MH and entering into another cell is referred to as a handoff. The process of handoff is responsible for maintaining end-to-end data movement connectivity and is transparent to the MH. A DBS provides full database services and it communicates with MHs only through a BS. DBSs can either be installed at BSs or can be a part of FHs or can be independent to BS or FH. MHs are mobile processors with some light weight database functionality such as requesting locks, executing transactions but can not terminate (complete) the transaction (can not change the final database state).

3 Review of earlier works

In this section, we review some commonly used CCMs and a variety of MT processing model for managing database consistency for improving performance.

3.1 Concurrency control mechanisms

Commonly used CCMs are based on locking approach [7, 19]. A variety of locking-based CCMs have been developed to cater for high contention transaction traffic. Most of these algorithms are resource intensive and they would not provide acceptable performance if used in MDS. In addition, these algorithms would not be able to manage long and short-lived MTs satisfactorily. Such situations do arise in conventional systems as well; however, on MDS, its effect on system performance will be comparatively much more significant.

In optimistic schemes [21], 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 before the commitment of MTs. This scheme leads to aborts of MTs unless the conflicts are rare. Since MTs are expected to be long-lived they may suffer higher degree of conflict where as our objective is to have very low abort rate in high conflicting traffic.

In pessimistic schemes [11], cached objects can be locked exclusively and MT can be committed locally. Another optimistic method [18] minimizes the overheads and transaction abort ratio by executing transactions only locally using caches. These schemes lead to unnecessary transaction blocking since mobile hosts cannot release any cached objects while it is disconnected. Existing caching methods attempt to cache the entire data object or the complete file. Caching of these potentially large objects over low bandwidth communication channels can result in wireless network congestion and high communication cost and the limited cache size of MH is unable to cache all desired data.

3.2 Mobile transaction execution models

Researchers have investigated different ways of structuring and processing MTs to improve performance and system availability. These processing approaches are not directly related to concurrency control so we review them briefly.

Semantic based transaction processing models [9, 37] have been extended for mobile computing in [45] to increase concurrency by exploiting commutativity of operations. These techniques require caching large portion of the database or maintain multiple copies of many data items. In [45], fragmentation of data objects has been used to facilitate semantic based transaction processing in MDS. The scheme fragments data objects where each fragmented data object has to be cached independently and manipulated synchronously. It, however, works only in situations where the data objects can be fragmented like sets, aggregates, stacks and queues.

Dynamic object clustering has been proposed in mobile computing in [31, 32] 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 [7]. 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 "commit" and "global commit" is same as our "termination" (see Sect. 4). However, a weak transaction after local commit can be aborted and/or compensated. In our model, a committed transaction does not abort and therefore, no undo or compensation is required. A weak transaction's updates are visible to other weak transactions whereas pre-writes are visible to all transactions. Reference [26] presents a new transaction model using isolation-only transactions (IOT) which do not provide failure atomicity and are similar to weak transactions of [31].

An open nested transaction model has been proposed in [10] for modeling mobile transactions. The model allows MT to be executed in disconnected mode. It supports unilateral commitment of compensating and sub-transactions. However, not all operations are compensated [10]. A Kangaroo Transaction (KT) model is presented in [14]. It incorporates the property that transactions in MDS hop from one base station to another as the mobile unit moves. The mobility of MT is captured by the use of split transaction [36]. A split transaction divides an on-going transaction into serializable subtransactions. Earlier created subtransaction may commit and the next subtransaction can continue its execution. The mobile transaction splits when a hop occurs. The model captures the data behavior of the mobile transaction using global and local transactions. The model also relies on compensating transaction in case a transaction aborts. Unlike KT, our model does not need any compensatory transaction and has low abort-rate. [38, 39] presented a speculative transaction processing approach, which can increase concurrency, but require more resources since it has high abort-rate in case of conflicts.

Transaction models for mobile computing that perform updates at mobile computers have been developed in [10, 31]. They propose a new correctness criterion [10] that is weaker than serializability. They can cope more efficiently with the restrictions of mobile and wireless communications.

In [27], a *prewrite* operation is used before a write operation in a mobile transaction to improve data availability. A prewrite operation does not update the data object but only makes visible the future value 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). The remaining transaction's execution is shifted to DBS. Writes on database consume resources at stationary host and are therefore, delayed. A pre-committed transaction's prewritten value is made visible both at mobile and stationary hosts before the final commit of the transaction. Thus, increases data availability during frequent disconnection common in mobile computing. The model also provides only serializable schedules [29].

In [20] a mobile transaction model called *Mobilaction* was presented. In *Mobilaction,* a location property was added in the ACID properties of conventional transaction to manage location dependent processing, which is inherent in MDS. This work also presented a commit protocol for *Mobilaction* and discussed its performance, which showed that the model successfully handles location dependent queries.

Our motivation is to increase availability with efficient use of limited channels. It is well-known that multiversion schemes significantly improve concurrency and many such algorithms [7, 17, 24] use bounded number of versions to improve system performance. The mixed multiversions [3, 6, 12, 44] have two types of transactions, i.e., the read-only and update transaction. The read operation reads the old but consistent versions while update (write) generates a new version of the old consistent version. We believe that multiversion approach may work well in MDS if reads are allowed only at mobile host and final writes are executed at DBSs. Multiversion schemes using two phase locking [7, 22] utilize the versions to allow the concurrent execution of the conflicting transactions. Since the concurrent access of the conflicting reads and writes 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 of executing the transaction that fails validation is wasted and is undesirable in MDS because aborts due to failed validation grows rapidly and gets worse with transaction size. Agarwal and Krishnamurthy [2] have used multiversion concurrency control for write-only transactions (blind-writes) and multiversions of objects have been proposed in disconnected databases [33], where transactions can work locally, but they are aborted if they get involved in conflicts.

In MDS, therefore, a mobile transaction processing scheme and the concurrency control mechanism must be able to achieve a balanced execution in the presence of limited resources and autonomy of the mobile host with respect to read and write operations. We first present our transaction-processing model and then discuss the locking based concurrency control mechanism.

4 A mobile transaction processing model

We discuss a multiversion transaction (MV-T) execution scheme, which increases data availability and reduces abort-rate using data versions. A successful MT in MV-T goes through three states (a) start, (b) commit, and (c) terminate. At any time an active MT may exist in any one of these states. A MT can start and *commit* (different than usual commit in DBMS) at a mobile host (MH) but it *terminates* only at one of the database servers (DBS). Our scheme synchronizes read and write lock requests on different versions of a data item in a constrained manner. The constraints are specified in terms of timestamps on the lock requested and on the lock held for the data item

(Sect. 6). The correctness of the transaction execution is guaranteed if the transaction can announce its commit by submitting its commit action to the server. No separate validation phase is required. The model supports concurrent read and write operations without blocking. A Read always gets the last committed or terminated version and is never blocked. MV-T scheme increases data availability at MH and at the server and supports both short and long transactions without very high block-rate or aborting short-transactions.

We explain MV-T scheme through the execution of MTs. A MT arrives at MH where it executes partially and the completion phase is moved to a DBS for execution. We consider the following scenario: A MT can arrive at a DBS from any MH and at the same time other MTs can start at DBSs initiated by other hosts connected via fixed network.

We assume that a MH starts and initiates the *commit* of a new MT but it always *terminates* at DBS. The ***commit of the transaction*** is the logical completion of the transaction after which MH sends the updated data items to DBS. The ***termination of a transaction*** is the state at which DBS revokes all the locks assigned to the transaction and the modified data items are successfully installed in the database at DBS. A transaction that committed successfully at MH is assured of successful termination at DBS. Note that MH has no DBMS capability to finish the transaction execution. It can only create a new value or image of a data item, but only DBMS can install that in the database.

In our execution model, we improve the concurrency of mobile transactions by making use of the time between the commitment of the mobile transaction at MH and the termination of the transaction at DBS. Consider the following examples and applications.

Example Consider two transactions T1 and T2 being executed at two MHs controlled by the same DBS (Fig. 2). Transaction T2 starts as soon as T1 commits but before it terminates. T2 does not wait for the transaction T1 to terminate in order to start its operations, thus increases concurrency. If a transaction is allowed to commit at MH, the data item values that are written by the transaction at MH, are send to DBS, which are then available to other transactions.

Application 1 In a mobile banking application where a bank agent can visit different regions to collect deposits/withdrawn of money in remote zones using a mobile de-

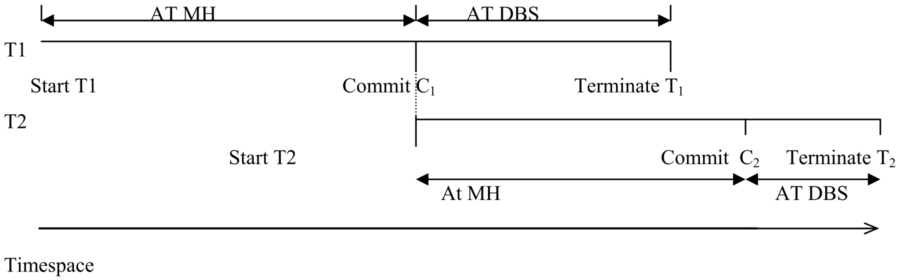


Fig. 2 Concurrent transaction execution

vice, once a check-deposit transaction is executed, the balance gets updated (commit) but the transaction *terminates* only when the money is deposited in the bank which then can be withdrawn. However, the account balance is available for reading after the commit of the check-deposit transaction at the mobile device.

Application 2 In emerging wireless networks, a MH can broadcast in its Cell committed values along with the time stamp and therefore, can save substantial wireless communication overhead as other MHs may not have to send their read requests to DBS for read-only locks and can always compare the broadcasted values to get the most recent updated values. It is only when a MH wants to read and write, locks need to be set. This increases the availability and reduces response time for read-only data objects.

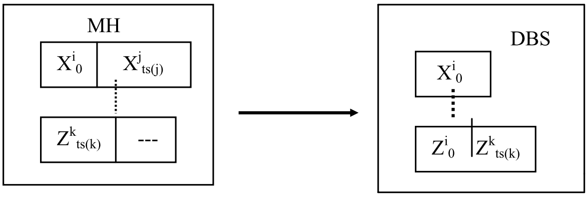
4.1 Multiversion of data items

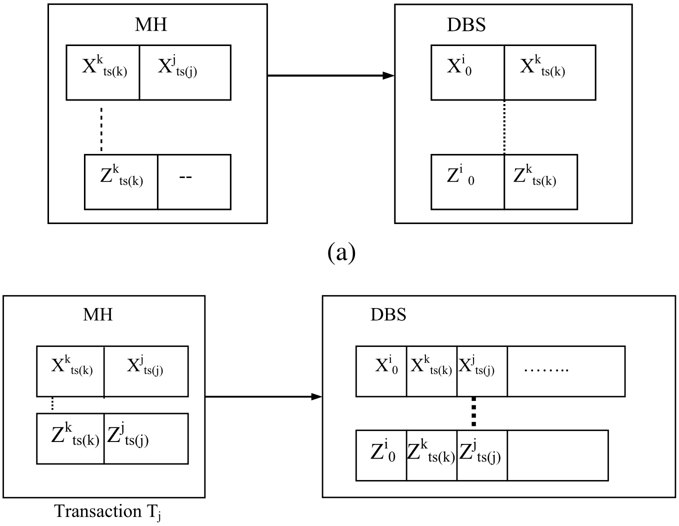
We use the concept of two versions (one committed and one terminated) to develop our transaction execution model. We then extend our scheme for multiversion (many committed and one terminated) case. Initially, we maintain two versions of a data item at any particular instance. When a MH requests for a data item, one of the two versions, depending upon the specified constraints (discussed later in the section), is granted. We represent a version of data item as $X^i_{ts(i)}$, where 'X' is the data item; '$ts(i)$' is the timestamp of the mobile transaction T_i that has written the version of the data item. '$ts(i)$' stands for the current timestamp of data item version $X^i_{ts(i)}$ used in version selection to process a read operation on 'X'. It also implies that the transaction T_i, which has updated the data item, has been successfully committed at MH but is yet to be terminated at DBS. DBS assigns timestamps to the data items when a transaction accesses them, and are assumed to be synchronized across the system.

Formally, the two versions of a data item 'X' maintained at DBS are: X^j_0 and $X^k_{ts(k)}$. X^j_0 is the data version written by the mobile transaction T_j, which has been terminated successfully at DBS. The subscript 'zero' indicates the successful termination of T_j at DBS. $X^k_{ts(k)}$, as said earlier, is the new version of data item created by the committed mobile transaction T_k at MH that has created the new value at time $ts(k)$, but is not yet terminated ($T_j < T_k$ in timespace) at DBS. We discuss two cases.

Case 1: *Concurrent read-write access to increase availability*: In this case, we discuss how committed and terminated versions are used to increase the concurrency among read and write operations. In Fig. 3, DBS has one version of data item 'X' and two versions of data item 'Z'. Versions, X^i_0 and Z^i_0, represent that a transaction T_i which updated X and Z has been most recently terminated. The version $Z^k_{ts(k)}$, indicates that there is a transaction T_k that is committed but is yet to be terminated. The data item versions present at MH indicate that earlier the transaction T_j being executed at MH has requested a read operation on data item 'Z' and a write operation on data item 'X'. DBS assigned the most up-to-date version of data items to the read operation. Hence, T_j read data item versions $Z^k_{ts(k)}$ (committed version) and X^i_0 (terminated version) available at DBS. After obtaining write permission (write-lock) on data item 'X', it writes its version $X^j_{ts(j)}$. Version $Z^k_{ts(k)}$ at DBS indicates that

Fig. 3 Case 1: data versions at MH and DBS





(a)

(b)

Fig. 4 **a** Case 2: data versions at MH and DBS, before T_j commits at MH. **b** Data versions at DBS and MH (case 2) after T_j commits at MH

there existed a transaction T_k that has committed at MH but not yet terminated. In our model, in order to maintain exactly two versions of a data item, we do not assign write permission on a data item to any transaction if there is another transaction that holds the write permission on that data item. The concurrent write operations on a data item conflict. Also, a transaction abort after acquiring write permission might result in cascading aborts. It is also possible that both transactions can simultaneously commit at MH resulting in the existence of more than one committed version of data item. Therefore, MH cannot obtain a write permission and write its version of 'Z' but can only read the version $Z^k_{ts(k)}$.

Case 2: *Concurrent write-write access to increase availability:* In Fig. 4(a), we see that there exist two versions of data items 'X' and 'Z' at DBS. As discussed earlier, the data items $X^k_{ts(k)}$ and $Z^k_{ts(k)}$ represent that there exist a committed but yet to be terminated transaction T_k. The data items X^i_0 and Z^i_0 represent a transaction T_i that is most recently terminated at DBS. We see that the transaction T_j being executed on MH reads the version $X^k_{ts(k)}$ at MH. Here, we have relaxed the condition that: "There cannot be more than one committed version of data item at DBS". That is, DBS assigns the write permission to transaction T_j at MH on data item 'X' even though there exists a committed but yet to be terminated transaction T_k at DBS that has a committed version $X^k_{ts(k)}$. This is possible because the transaction T_k is committed and therefore assured of successful termination. The versions of a data item written by such transaction are up-to-date and can be used by another transaction. Since transactions can be terminated only at DBS, it can keep track of the order in which transactions need to be terminated. Thus, DBS can give a write permission on a data item when there exists another transaction that is committed on the same data item but yet to be terminated. Hence, T_j at MH is assigned the write permission on data item 'X' with version $X^k_{ts(k)}$. It then writes its own version of data item, $X^j_{ts(j)}$. The transaction T_j then commits at MH and sends the version $X^j_{ts(j)}$ to DBS to terminate.

Though our discussions in this paper is focused on two versions, however, in general, we can see that there can be more than one committed versions of a data item present at DBS resulting from transactions that have written newer versions of data item 'X' and are successfully committed at MH but yet to be terminated at DBS (Fig. 4(b)). DBS terminates them in the order in which they have been committed earlier. In Fig. 4(b), T_k terminates first and T_j is terminated next and so on. Thus, DBS improves concurrency. Note that if a new transaction T_l arrives at DBS requesting read or write operation on the data item 'X' when T_j holds the write access, it is blocked until T_j commits.

The actual lock-acquiring scenario is discussed in the next section where we introduce the locking protocol and read and write constraints.

5 Locking protocol

We have used locking for achieving isolation, timestamp to avoid deadlock, and use incremental locking to eliminate cascading aborts. We introduce an additional lock type called verified-lock. A write-lock is converted to a verified-lock after a MT's commit. If there are conflicting lock requests then MTs are either blocked or aborted. We refer to a MT that is requesting a data item as *requestor* and the MT that holds the requested data as the *holder*. A requestor is blocked when its timestamp is higher than the holder.

A MT gets the latest version of data item on a read request. This defines our *Read Rule*, which is applied to all read requests. Note that in an optimistic model a transaction usually reads an older version of the data item. A requestor is blocked when it does not get the data item. Action taken by the scheduler on the lock request that fails to satisfy the constraints is rejected to avoid conflicts. Since no MT is blocked indefinitely, there is no deadlock. We discuss the following two cases to explain how locking is applied to them.

Locking rules for case 1 DBS assigns the locks to the requesting MTs. These transactions can be initiated at a MH or at DBS. There are two kinds of read locks: $rl^{=0}(X)$ and $rl^{\neq 0}(X)$. These read-locks differentiate two versions of data items: $rl^{=0}(X)$ for terminated version X^i_0 and $rl^{\neq 0}(X)$ for committed version $X^k_{ts(k)}$ respectively. We have write lock $wl(X)$ for write operation and a verified lock $vl(X)$ which shows the

T_i holding lock at DBS

	Readlock(rl)	Writelock(wl)	Verifiedlock (vl)
Read lock (rl)	√	√	√
Write lock(wl)	√	X	X
Verified lock(vl)	√	X	X

T_j requesting locks at MH

Fig. 5 Lock compatibility matrix (for case 1 only)

transition from 'Commit State' of mobile transaction at MH to 'Termination State' of mobile transaction at DBS.

A MT at MH acquires the required locks on data items before performing any read or write. DBS assigns MH the appropriate version of the data item to read. The requested lock on a data item is assigned in such a manner that there is no other MT holding a conflicting lock. The read locks do not conflict with any of the read, write or the verified lock. The write lock conflict with write and verified locks because a MT can be aborted after it has acquired the write lock and if some other MT is assigned the write lock on the same data item, it can result in cascading aborts. One may think that both MTs can simultaneously commit at MH and convert their write locks to verified locks which may result in the existence of more than one committed version of data item (possible only when the other MT holding write lock has already committed as discussed earlier in Case 2, Fig. 4(b)). This situation may occur when both MTs try to obtain a verified lock, which results in violation of the condition—*there exists at the most two versions of data item at any instance of time at a DBS. One of which is a terminated version and the other is committed but yet to be terminated version of data item.* The lock compatibility matrix shown is in Fig. 5.

There are two constraints that must be satisfied by any MT for obtaining locks. DBS checks constraint before assigning locks and assigns verified lock to a MT when it has completed all its reads and writes and has committed. The actions performed when a MH executes commit are:

i. The new versions of a data item (if any) are sent to DBS.
ii. The write locks held by MTs are converted into verified locks at DBS.
iii. The committed version of data items written by MT, $X^i_{ts(i)}$ is available for other MTs.

DBS later executes a terminate 't_i' command to end MT's execution. The different actions performed at the execution of terminate command at DBS are:

i. All MTs holding read locks $rl^{\neq 0}(X)$ on data item version $X^i_{ts(i)}$ are converted to $rl^{=0}(X)$.
ii. The previous committed and terminated versions of data item X^j_0 are deleted and data version $X^i_{ts(i)}$ is converted to X^i_0.
iii. All the verified locks are revoked and so are the read locks assigned to T_i.
iv. Once locks are revoked and the version of the data item is being updated at DBS, it completes the MT.

T_i holding lock at DBS

	Readlock(rl)	Writelock(wl)	Verifiedlock (vl)
Read lock (rl)	√	√	√
Write lock(wl)	√	X	√ (T_i-Committed)
Verified lock(vl)	√	X	√ (T_i- Committed)

T_j requesting locks at MH

Fig. 6 Lock compatibility matrix (case 2)

Due to the conflict between write and verified locks only two versions of a data are available at DBS. The compatibility matrix in Fig. 6 shows that no two MTs can have write locks on the same data item simultaneously.

Locking rules for case 2 In this case we relax the constraint that *at most only two versions of data item at any instance of time* are available to improve concurrency. The compatibility matrix for this case is shown in Fig. 6 and the data versions scenario at DBS and MH is shown in Fig. 4(b). In Fig. 6, if a MT holding the verified lock on the data item is committed but is still to be terminated then it can assign the conflicting write lock or verified lock to another MT at MH. It can do that by maintaining the order in which MTs need to be terminated, as discussed earlier in Case 2 (Fig. 4(b)). This is possible because DBS can only assign locks to MH and terminates a transaction. Thus, when T_j at MH requests DBS a write or a verified lock on data item 'X', DBS can assign the write lock or verified lock to T_j provided the current lock held by T_i on DBS is the verified lock. The DBS records the order of verified locks after assigning the lock to T_j. When T_j commits, DBS performs termination in the order in which it has assigned the verified locks, i.e., T_i and then T_j. If T_i cannot be terminated, then T_j is held from termination until T_i is terminated. In this way DBS preserves the correct serial order of MT execution. Also, the value read by T_j is a correct value written by T_i since T_j gets the write lock only after T_i is committed but is yet to be terminated.

In MDS, a MH can encounter handoff randomly. This gets worse with highly mobile MHs. We argue that a handoff does not affect our locking protocol. If a lock is not granted to a transaction then it is blocked irrespective of its connection status and its geographical location. For example if a transaction running at a MH requests a lock in cell 1 and the MH moves to cell 2 then irrespective of its geographical location, the transaction will be blocked. Note that the change in the status of a transaction (blocked or aborted) is free from the movement and status of the MH where it is executing. Of course the movement (handoff) will have some effect on the behavior of our schemes which is included in the performance study (we assume 10% handoff in all our experiments). Further, it is useful to investigate the situation when a disconnection occurs before the MH receives a lock granted confirmation. This can be treated as "not granted" if the disconnection is flowed with "doze mode". We have modeled the effect of disconnection after a handoff in this manner.

The status of the underlying network (i.e., network partition) does have some effect on transaction management. This, however, is true only in wired network. In cellular network on which MDS is mounted, such network partition does not exist. If a MH crosses the geographical mobility domain (Fig. 1), then it can be treated as a disconnection and our scheme is capable of handling this.

6 Constraints with read and write operations

In our model, a read request is completed at DBS using a Read rule similar to the multi-version timestamp ordering (MVTO) read rule [7]. Whenever a transaction wants to read a data item then the committed version of the data item with the largest timestamp less than or equal to the timestamp of the transaction is selected. That is, if there exist versions: X_0^1 and $X_{ts(2)}^2$ at DBS and transaction $T3$ at MH or at DBS requesting data item X having a timestamp $ts(T3) > ts(T2)$, then $T3$ is given $X_{ts(2)}^2$ and otherwise X_0^1. A read protocol on is stated as follows:

- T_i requests a read lock on the data item X.
- DBS grants $rl_i^0(X)$ or $rl_i^{\neq 0}(X)$ corresponding to whether the version X_0^j or version $X_{ts(k)}^k$ (if it exists and is committed) is selected in accordance with the read rule; and the read lock version satisfies the specified constraints.
- Transaction T_i reads the selected version of X.

The write protocol is stated as follows:

- T_i requests a write lock on data item X.
- DBS grants $wl_i(X)$, if there are no conflicts.
- T_i creates a new version $X_{ts(i)}^i$ for data item X.

The following two constraints must be satisfied before the requested lock is granted:

Constraint 1 *If a transaction T_j at DBS holding $wl_j(X)$ lock, then transaction T_i gets read lock if timestamp$(T_i) >$ timestamp(T_j).*

This condition checks for the situation that no read request is processed violating the Read rule, that is, if the transaction T_j that is holding a write lock on the data item and has a timestamp less than the transaction T_i that is requesting a read lock then granting a read lock might result in violation of serializability and also might result in cascading aborts. This constraint satisfaction results automatically in maintaining serializability of transaction execution at DBS.

Constraint 2 *The write lock request $wl_i(X)$ or verified lock request $vl_i(X)$ for transaction T_i (at MH or at DBS) must satisfy*:

(a) *There does not exist any transaction at DBS holding $wl(X)$ or $vl(X)$ (verified lock); and for all transaction T_j at MH that holds $rl_j^0(X)$, the timestamp$(T_i) \geq$ timestamp(T_j).*

(b) *If there is any other transaction T_k at DBS holding $vl_k(X)$ lock, then timestamp$(T_k) <$ timestamp(T_i), where T_i is a requesting verified or write-lock on X.*

This constraint ensures that the transactions at DBS, already having the read locks on previous version of data items are not made void by assigning a write lock or verified lock to another transaction that comes after these transactions, thus avoiding aborts.

Rules for terminating a transaction The termination for T_i may not be invoked immediately after it commits. The following rules must be observed for correct execution of transactions:

1. T_i at MH will precede T_j at DBS in commit order if T_i has read a previous version of a data item for which T_j has created a new version or T_j has read the committed version of the data item written by T_i. This is because if T_i reads the previous version of a data object, which has later been updated by T_j then if T_j commits before T_i then it should have read the updated version. Note that a read-only transaction also needs to send the commit information to DBS. Other alternative is that such read-only transaction can be switched back in the transaction history for the serialization purpose. In the second possibility, if T_j has read the committed version written by T_i then T_i should come before T_j in the serialization order.
2. T_j can not terminate at DBS until each transaction T_i at MH that has either read X_0^k (for some k) or written a committed version $X_{ts(i)}^i$ that has been read by T_j, has been terminated. This is because a transaction may be reading two data items, for one it may get a data version written by the last terminated transaction and for another data item version it may read, is written by the committed transaction. Thus, there is no equivalent serial order as read-only transaction read one version of data object at initial state and updated version of the second object.
3. T_i executed at MH cannot terminate at DBS until T_j that has committed before $T_{i,}$ terminates at DBS.

Blocking transactions Deadlocks and subsequent aborts could be costlier in a mobile environment. Also, wireless connection from MH to DBS is expensive; therefore, we would like to avoid MH contacting DBS as far as possible. In some situations when a lock cannot be granted, a transaction can be blocked rather than aborted and when the lock is available, it can be unicasted. Consider a case where a read transaction initiated by MH does not satisfy constraint-1 then it should be blocked at DBS rather than aborting it. When at DBS the write lock of a transaction is converted into the verified-lock, the transaction blocked at MH can read the committed version. DBS can unicast this message and MH needs not be contacted DBS again.

Deadlock avoidance rule If T_i holds a write- or verified-lock lock on X then the write-lock request on X by other transaction T_j is rejected if $\mathit{timestamp}(T_i) > \mathit{timestamp}(T_j)$; otherwise T_j is blocked.

Using the above rule along with the write-lock request that fail the constraint 2 can make the execution deadlock free. Since lock requests are blocked in asymmetric fashion; only transaction with higher timestamp may be blocked by a transaction with a lower timestamp, there will not be any deadlocks.

6.1 Comparison with constrained shared locking model

The lock acquisition in our proposed model has some similarity with the constrained shared locking model in [1]. The Lock Acquisition rule in constrained shared locking model states that: in an history H, for any two operations $p_i[x]$ and $q_j[x]$ such

T$_i$ holding lock at DBS

	Readlock(rl)	Writelock(wl)	Verifiedlock (vl)
Readlock (rl)	√	√	√
Writelock(wl)	√	X	⇒ (T$_i$-Committed)
Verifiedlock(vl)	√	X	⇒ (T$_i$- Committed)

T$_j$ Requesting at MH

Fig. 7 Lock compatibility matrix (case 2)

that $pl_i[x] \Rightarrow ql_j[x]$ is permitted, if T_i acquires $pl_i[x]$ before T_j acquires $ql_j[x]$, then execution of $p_i[x]$ must precede the execution of $q_j[x]$. In our proposed model, according to Property$_{2a}$: For any two transactions T_i and T_j at DBS, if $c_i < c_j$ then $vl_i(x) < vl_j(x)$ and $t_i < t_j$, we state that the transactions obtain verified locks at DBS and also terminate in the order they commit. That is, for two transactions T_i and T_j if there exists an ordering $c_i(x) \Rightarrow c_j(x)$ then

1. they obtain verified locks in the same order of the form $vl_i(x) \Rightarrow vl_j(x)$ and
2. the corresponding termination operations have ordering of the form $t_i \Rightarrow t_j$.

There is an ordering for obtaining conflicting write locks and verified locks given a condition that one of the two transactions is a committed transaction at DBS. That is for a transaction T_i which is committed at DBS and holding $vl_i(x)$ lock, if there is another transaction T_j requesting a write-lock on x, it is assigned the write lock by maintaining an order between $vl_i(x)$ and $wl_j(x)$. Therefore if $vl_i(x) \Rightarrow wl_j(x)$ then $vl_i(x) \Rightarrow vl_j(x)$ and $t_i \Rightarrow t_j$.

The compatibility matrix shown in Fig. 6 is redrawn in Fig. 7 depicting the above-discussed cases where there is some similarity between the proposed model and the constrained shared locking model of [1].

7 Formal proof of correctness

A read operation executed by a mobile transaction T_i on a data object 'x' is denoted either as $r_i(x^k_{ts(k)})$ or $r_i(x^j_0)$ depending on the read and write rule constraints specified in Sect. 5. A write operation is executed as $w_i(x^j_{ts(j)})$. The commit is denoted as 'c_i', an abort as 'a_i' and terminate as t_i. When a transaction is committed its changes are updated at the DBS (DBS) and if it aborts, all the data versions that the transaction has created will be discarded. A transaction is correct if it maps the database at DBS (DBS) from one consistent state to another consistent state. Formally a mobile transaction T_i is a partial order with ordering $<_i$ where

i. $T_i \subseteq \{r_i(x^k_{ts(k)})$ or $r_i(x^j_0)$, $w_i(x^i_{ts(i)}) \mid x$ is a design object $\} \cup \{c_i, a_i, t_i\}$.
ii. If $r_i(x^k_{ts(k)}) \in T_i$ if and only if $r_i(x^j_0) \notin T_i$.
iii. If $a_i \in T_i$ if and only if $c_i \notin T_i$ and vice-versa.
iv. If 't' is c_i or a_i then for any operation $p <_i t$.
v. If $r_i(x^k_{ts(k)})$, $w_i(x^i_{ts(i)}) \in T_i$ then $r_i(x^k_{ts(k)}) <_i w_i(x^i_i)$.

Let $T = \{T_0, T_1, T_2, \dots, T_n\}$ be a set of transactions, where the operations of T_i are ordered by $<_i$ for $0 \le i \le n$. To process operations from T, a multiversion scheduler must translate T's operations on (single version) data items into operations on specific versions of those data items. This translation can be formalized by a function "f" that maps each $w_i(x)$ into $w_i(x^i_{ts(i)})$, each $r_i(x)$ into $r_i(x^k_{ts(k)})$ for some k, each c_i into c_i, and each a_i into a_i.

A *complete multiversion* (*MV*) *history* [7] H over T is partial order with order relation $<_H$ where

i. $H = f(\bigcup_{i=0}^{n} T_i)$ for some translation function 'f', which is the combination of read and write rule constraints in our transaction model;
ii. for each T_i and all operations p_i and q_i in T_i, if $p_i <_i q_i$, then $f(p_i) < f(q_i)$;
iii. if $f(r_j(x)) = r_j(x^i_{ts(i)})$, then $w_i(x^i_{ts(i)}) <_H r_j(x^i_{ts(i)})$;
iv. if $w_i(x) <_i r_i(x)$, then $f(r_i(x)) = r_i(x^i_{ts(i)})$; and
v. if $f(r_j(x)) = r_j(x^i_{ts(i)})$, $i \neq j$, and $c_j \in H$, then $c_i <_H c_j$.

A *committed projection* of a MV history H, denoted as $C(H)$, is obtained by removing from H the operations of all but the committed transactions. $C(H)$ is *complete MV history* if H is a MV history [7]. Two MV histories over a set of transactions are equivalent iff the histories have the same operations. Two operations in an MV history *conflict* if they operate on the same version and one is a Write. Only one pattern of conflict is possible in an MV history: if $p_i < q_j$ and these operations conflict, then p_i is $w_i[x^i_{ts(i)}]$ and q_j is $r_j[x^i_{ts(i)}]$ for some data item x. The other type of conflicts [7] are not possible. Thus conflicts in MV history correspond to reads-from relationships.

A complete MV history is *serial* if for a pair of transactions T_i and $T_j \in H$, either all operations executed by T_i precede all operations executed by T_j or vice versa. Not all serial MV histories behave like ordinary serial 1V histories [7]. A serial MV history H is *one-copy* (or 1-serial) if for all i, j, and x, if T_i reads x from T_j, then $i = j$, or T_j is the *last* transaction preceding T_i that writes into any version of x. A MV history is *one-copy serializable* (or 1SR) if its committed projection, $C(H)$, is equivalent to an 1-*serial* MV history.

The serialization graph SG(H) for an MV history is defined as for an 1V history. Given an MV history H and a data item x, a *version order*, $\ll$, for an x in H is a total order of versions of x in H. A *version order* for H is the union of version orders of data items. For example for a history $H = w_0[x^0_{ts(0)}]$, $w_0[y^0_{ts(0)}]$, $w_0[z^0_{ts(0)}]$, $r_1[x^0_{ts(0)}]$, $r_2[x^0_{ts(0)}]$, $r_2[z^0_{ts(0)}]$, $r_3[z^0_{ts(0)}]$, $w_1[y^1_{ts(1)}]$, $w_2[x^2_{ts(2)}]$, $w_3[y^3_{ts(3)}]$, $w_3[z^3_{ts(3)}]$, c_1, c_2, c_3 possible version orders are $x^0_{ts(0)} \ll x^2_{ts(2)}$, $y^0_{ts(0)} \ll y^1_{ts(1)} \ll y^3_{ts(3)}$.

The *Multiversion Serialization Graph* for H and version order $\ll$, MVSG($H, \ll$), is SG(H) with the following version order edges added: for each $r_k[x^j_{ts(j)}]$ and $w_i[x^i_{ts(i)}]$ in $C(H)$ where i, j and k are distinct, if $x^i_{ts(i)} \ll x^j_{ts(j)}$, then include $T_i \rightarrow T_j$ otherwise include $T_k \rightarrow T_i$. An MV history H is 1SR and hence conflict serializable, iff there exists a version order $\ll$ such that MVSG($H, \ll$) is acyclic.

We now discuss a few possible histories during concurrent transaction executions according to our model. We assume that the multiversion scheduler of our transaction model starts in an initial correct and consistent database state D_0, with a single version x^0_0 for each data item x^k_j in the database

1. The history produced by our transaction is conflict serializable and is equal to serial 1V history.

Consider the following simple history that conforms to both read and write rule constraints during obtaining locks. Given $ts(1) < ts(2) < ts(3) < ts(4)$.

$$rl_1^{=0}(x)r_1(x_0^0)rl_2^{=0}(x)r_2(x_0^0)c_2wl_1(x)w_1(x_{ts(1)}^1)c_1(wl_1(x) \rightarrow vl_1(x))rl_3^{\neq 0}(x)$$

$$r_3(x_{ts(1)}^1)wl_3(x)w_3(x_{ts(3)}^3)t_2c_3(wl_3(x) \rightarrow vl_3(x))rl_4^{\neq 0}(x)r_4(x_{ts(3)}^3)c_4t_1t_3t_4.$$

The MVSG for the above history is $T_0 \leftarrow T_2$ $T_1 \leftarrow T_3 \leftarrow T_4$ (assuming that the version x_0^0 results from an earlier terminated transaction T_0). It is acyclic. If we remove the versions of data items then that would also results in:

$$rl_1^{=0}(x)r_1(x)rl_2^{=0}(x)r_2(x)c_2wl_1(x)w_1(x)c_1(wl_1(x) \rightarrow vl_1(x))rl_3^{\neq 0}(x)$$

$$r_3(x)wl_3(x)w_3(x)t_2c_3(wl_3(x) \rightarrow vl_3(x))rl_4^{\neq 0}(x)r_4(x)c_4t_1t_3t_4.$$

It is clear that the above history is a serial 1V history with the execution order of $T_2 \rightarrow T_1 \rightarrow T_3 \rightarrow T_4$.

We now prove that if a transaction doesn't conform to either the read rule or the write rule constraint then it will not result in a serial 1V history.

2. A transaction cannot be assigned locks if there is some other uncommitted transaction at DBS holding conflicting locks.

$$rl_1^{=0}(x)r_1(x_0^0)rl_2^{=0}(x)r_2(x_0^0)c_2wl_1(x)w_1(x_{ts(1)}^1)rl_3^{\neq 0}(x)r_3(x_{ts(1)}^1)c_1(wl_1(x)$$

$$\rightarrow vl_1(x))rl_3^{\neq 0}(x)r_3(x_{ts(1)}^1)wl_3(x)w_3(x_{ts(3)}^3)t_2c_3(wl_3(x)$$

$$\rightarrow vl_3(x))rl_4^{\neq 0}(x)r_4(x_{ts(3)}^3)c_4t_1t_3t_4.$$

The above history is not possible since transaction T_1 holds write locks and hence the read lock $rl_3^{\neq 0}(x)$ cannot be assigned to T_3 against the read rule constraint.

Properties We now discuss possible properties that every operation in the transaction execution history of our mobile transaction model conforms to, and then later will prove the correctness of our locking protocol scheme by describing it using multiversion serializability theory discussed earlier in the section[1] and confirming that all histories produced by it are 1SR.

Let H be a history over T $\{T_1, T_2, T_3, \ldots\}$ produced by our locking protocol. Then H must satisfy the following properties.

Property 1 For each T_i, there is a unique timestamp $ts(T_i)$. For simplicity, we assume that $ts(T_i) = ts(i)$.

Property 2 For each T_i, the terminate action t_i follows after the commit action and the verified lock $vl_i(x)$ acquisition; i.e. $c_i < vl_i(x) < t_i$.

[1] Interested users can refer to reference book [7] for a detail discussion on multiversion serializability theory.

Property 2a For any two transactions T_i and T_j at DBS, if $c_i < c_j$ then $vl_i(x) < vl_j(x)$ and $t_i < t_j$.

Property 2a states that if two transactions commit in an order they acquire and hold verified locks at DBS in the same order as they commit and they are terminated in the same order as they had acquired the verified locks, thus maintaining the serializability of transaction execution and allowing DBS to assign conflicting *vl and wl locks.*

Property 3a For each $r_k[x_0^j] \in H$, either (1) $t_j < r_k[x_0^j]$ and $j > 0$; or (2) $x_0^0 \in D_0$ (initial consistent state of database of data items at DBS).

Property 3b For each $r_k[x_{ts(j)}^j] \in H$, either (1) $c_j < r_k[x_{ts(j)}^j] < vl_j(x) < t_j < vl_k(x) < t_k$ and $ts(x_{ts(j)}^j) < ts(T_k)$; or (2) $w_j[x_{ts(j)}^j] < r_k[x_{ts(j)}^j]$ and $j = k$.

Property 4 For each $r_k[x_{ts(a)}^l]$ and $w_k[x_{ts(k)}^k] \in H$; if $w_k[x_{ts(k)}^k] < r_k[x_{ts(a)}^l]$ then $a = k$ and $l = k$.

Property 3a, 3b together says that every Read $r_k[x]$ either reads a committed version or reads a version written by itself (i.e. T_k). In either case, it reads the version with the timestamp less than or equal to $ts(T_k)$. $t_j < t_k$ in Property 3b follows from the definition of unary relation *terminates*. Property 4 says that if T_k wrote x before the scheduler received $r_k[x]$, it translates the request to read the version written by T_k.

Property 5a For every $r_k[x_0^j]$ and $w_i[x_{ts(i)}^i] \in H$; either $t_i < r_k[x_0^j]$ or $r_k[x_0^j] < t_i$.

Property 5a says that $r_k[x_0^j]$, i.e. a read on the version x_0^j, created by the terminated transaction T_j at DBS, is strictly ordered with respect to the terminate action of every transaction (either at MH or at DBS) that writes x. This is because each transaction T_i *that writes* $x_{ts(i)}^i$ and commits at MH making the version available for other transactions holds a verified lock $vl_i(x)$ at DBS. Each such transaction T_i waits for each transaction that has read the existing version x_0^j to terminate, before it can terminate and release $vl_i(x)$ lock. Since the vl and wl locks conflict (according to case 1, Fig. 5) for each transaction T_k that reads x_0^j, either T_i must have terminated before T_j even got the $wl_j(x)$ lock, i.e. $t_i < wl_j(x) < t_j < r_k[x_0^j] < t_k$; or T_i must have terminated after T_k reading the version x_0^j had terminated, i.e. $r_k[x_0^j] < t_k < t_i$.

Property 5b For every $r_k[x_{ts(j)}^j]$ and $w_i[x_{ts(i)}^i] \in H$; if $w_i[x_{ts(i)}^i]$ then (1) either $vl_i(x) < r_k[x_{ts(j)}^j]$ or $t_i < r_k[x_{ts(j)}^j]$; else (2) $r_k[x_{ts(j)}^j] < vl_i(x)$, $t_k < t_i$ and $ts(k) < ts(i)$.

Property 5b says that $r_k[x_{ts(j)}^j]$, i.e. a read on a committed version $x_{ts(j)}^j$ due to a transaction T_j committed successfully at MH but yet to be terminated at DBS, is strictly ordered with respect to the every transaction that holds verified lock $vl_i(x)$ at DBS after committing at MH and writing a version of data item x.

(1) As discussed in case 2, Fig. 6, DBS assigns conflicting *vl* and *wl* locks on data items depending on the condition that the transactions that have written new version of data item must be committed successfully at MH and hold a verified lock at DBS in a strict complete order of execution. Therefore T_i must have either successfully committed at MH and have acquired $vl_i(x)$ or terminated and released the $vl_i(x)$ lock before T_j even got the $wl_j(x)$ lock, i.e. either $c_i < vl_i(x) < wl_j(x)$ or $t_i < wl_j(x) < c_j < r_k[x_j^j]$;

(2) By definition of the terminate action, t_j converts the version $x^j_{ts(j)}$ read by $r_k[x^j_{ts(j)}]$ into x^j_0; converts the $rl^{\neq 0}_k(x)$ lock into $rl^0_k(x)$ lock; and then releases the $vl_j(x)$ lock. By the Property 3b, $t_j < t_k$. Thus after T_j terminated and before T_k terminates, if $ts(k) > ts(i)$, $wl_i(x)$ lock request must wait for T_k to terminate and release the now $rl^0_k(x)$ lock in accordance with Constraint 2, i.e. $r_k[x^j_{ts(j)}] < t_j < t_k < wl_i(x) < t_i$; otherwise $ts(k) < ts(i)$ and T_i obtains the $wl_i(x)$ lock, writes the version $x^i_{ts(i)}$, and then waits for T_k that has read the new version x^j_0, to terminate, i.e. $r_k[x^j_{ts(j)}] < t_j < wl_i(x) < t_k < t_i$.

Property 6a For every $r_k[x^j_0]$ and $w_i[x^i_{ts(i)}]$ $(i, j, k$ distinct); if $t_i < r_k[x^j_0]$ then $t_i < t_j$.

Property 6b For every $r_k[x^j_{ts(j)}]$ and $w_i[x^i_{ts(i)}]$ $(i, j, k$ distinct); if $t_i < r_k[x^j_{ts(j)}]$ then $t_i < t_j$.

Property 6a says that $r_k[x^j_0]$ reads the most recently terminated version of x. Assume to the contrary that $t_j < t_i$. But then the version x^j_0 generated when T_j is terminated, must have been deleted and replaced by x^i_0 when T_i terminates, and thus $r_k[x]$ could not have accessed x^j_0. Property 6b combined with Property 3b says that $r_k[x^j_{ts(j)}]$ either reads the version written by itself or the most recently committed version $x^j_{ts(j)}$. Since the *vl* and *wl* locks conflict, if $t_i < r_k[x^j_{ts(j)}]$ then $t_i < w_j[x^j_{ts(j)}] < c_j$, which combined with Property 3b says $t_i < t_j$.

Property 7a For every $r_k[x^j_0]$ and $w_i[x^i_{ts(i)}]$, $i \neq j$, $j \neq k$, if $r_k[x^j_0] < t_i$ then $t_k < t_i$.

Property 7b For every $r_k[x^j_{ts(j)}]$ and $w_i[x^i_{ts(i)}]$, $i \neq j$, $j \neq k$, if $r_k[x^j_{ts(j)}] < t_i$ then $t_k < t_i$.

Property 7a, 7b says that T_i cannot terminate until every transaction that has read the existing terminated version, has terminated. Property 7a follows directly from the definition of unary relation *terminates*. Property 7b follows from Property 5b and Property 2a.

Property 8 For every $w_i[x^i_{ts(i)}]$ and $w_j[x^j_{ts(j)}]$, either $t_i < t_j$ or $t_j < t_i$.

Property 8 says that the termination of every two transaction that writes the same data item are atomic with respect to each other.

Theorem *Every history H produced is* $1SR$.

Proof By Property 2, Property 3a, 3b, Property 4, H preserves reflexive reads-from relationship and is recoverable. Hence it is a MV history. Define a version order $\ll$ as $x^i \ll x^j$ only if $t_i < t_j$. By Property 8, $\ll$ is indeed a version order. We will prove that all edges in MVSG($H, \ll$) are in the termination order. That is $T_i \to T_j$ in MVSG($H, \ll$) then $t_i < t_j$.

Let $T_i \to T_j$ be in SG(H). This edge corresponds to a reads-from relationship such as T_j reads x from T_i. By Property 3a $t_i < r_j[x_0^i]$ and from Property 2 $r_j[x_0^i] < t_j$. Hence $t_i < t_j$. Similarly, by Property 3b and Property 2a for any $r_j[x^i_{ts(i)}]$, $t_i < t_j$.

Consider a version order edge induced by $w_i[x^i_{ts(i)}]$, $w_j[x^j_{ts(j)}]$ and $r_k[x_0^j]$, (i, j, k distinct). There are two cases: $x^i \ll x^j$ or $x^j \ll x^i$. If $x^i \ll x^j$, then the version order edge is $T_i \to T_j$, and $t_i < t_j$ follows from the definition of $\ll$. If $x^j \ll x^i$, then the version order edge is $T_k \to T_i$. Since $x^j \ll x^i$, $t_j < t_i$ follows from the definition of the version order. By Property 5a either $t_i < r_k[x_0^j]$ or $r_k[x_0^j] < t_i$. In former case, Property 6a implies that $t_i < t_j$ contradicting $t_j < t_i$. Thus $r_k[x_0^j] < t_i$. By Property 7a, $t_k < t_i$ that is $T_k \to T_i$, as desired. The case of the version order edge induced by $w_i[x^i_{ts(i)}]$, $w_j[x^j_{ts(j)}]$ and $r_k[x^j_{ts(j)}]$, (i, j, k distinct) can be proved in exactly similar manner, just by applying Property 5b, Property 6b and Property 7b in place of Property 5a, Property 6a and Property 7a respectively as used in the above discussed case.

This proves that all edges in the MVSG($H, \ll$) are in termination order. Since termination order is embedded in a history, which is acyclic by definition, MVSG($H, \ll$) is acyclic too. Thus, from the definition of multiversion conflict serializable history, H is 1SR. □

8 Simulation model

First we study the behavior of our scheme with a simulation model and then compare their performance with SMTP model [38, 39] to establish better performance. This two level performance study was necessary to understand the behavior of our model and its comparison with a suitable reference model.

Table 1 lists the parameters and their values, which drive the simulator. The database size is represented in terms of lockable units and a small database size (DB_Size) is used to generate a high contention environment to show the resiliency of our scheme. In our architecture, therefore, we assume that transactions originate only at MUs and do not cache data. This assumption allowed us to measure the performance under high communication traffic. *Trans_size* is the average number of data objects required by a transaction, which is computed form their largest size (MAX_Objects) and smallest size (MIN_Objects). We assume that a read always precedes a write and *Write-Prob* defines this probability. The parameter *Transmission Delay* defines the

Table 1 Simulation parameters

Parameter	Description	Value
DB_Size	The number of data objects in the system	500/fixed
Num_MH	The number of mobile hosts in the system	125 (maximum)
MAX_Objects	Maximum number of data objects a transaction needs	20/Fixed
MIN_Objects	Minimum number of data objects a transaction needs	12/Fixed
Trans_Size	Average transaction size	17
MPL	Total number of active transactions in a MDS	5–125
Write-Prob	Probability that the object read will also be written	(0–1)
Object_Size	The size of the data object being read	250–500 kilobytes
Wireless-Bandwidth	Available transmission Bandwidth	5–10 kilobytes per ms
Transmission Delay	Message delay between a DBS and a MH	Message Size (Object_Size/ Bandwidth)
Read Time	Time taken to for reading an object	10 ms
Write Time	Time taken for writing a new value for the object	20 ms

communication delay between DBS and MH. This delay mainly depends on two parameters *Wireless-Bandwidth* available for transmission and *Object_Size* of the object being sent or received. *Read Time and Write Time* are the times required for reading an object from the database and writing a new value for the object in the database respectively. Most of these parameters have been used in the papers including [38, 39].

We have assumed that at any time there could be a maximum of 120 MHs in a cell moving (0 to 10 m/s) using Random Waypoint Model from which the transactions are initiated. Since we conducted our experiment with one cell only; we have used multiple MHs, one BS, one DBS, as BSs and DBSs are assumed to be connected with high bandwidth faster wired network, therefore, hand-offs are simulated in terms of delays (between 100–200 ms [43]) with 10% random hand-offs in each case. We consider random disconnections causing aborts in all our experiments with variable bandwidths. The effect was taken into account in all our experiments, which was not significant in terms of overall transaction response time.

8.1 Simulation results and discussion

The important parameters to define the performance of a concurrency control mechanism are *throughput, transaction abort, blocking* and *rates, response time,* and *Conflict rate* with in a range of multiprogramming levels (MPL) under a normal to stress situation. To minimize the stray effects of simulation we executed 5000 transactions for each simulation run. We executed the same run multiple times and took the average of the results. We define throughput as the number of transactions committed per unit time (milliseconds) and the response time as the total time spent by a transaction in the system. The restart rate defines how many transactions were restarted per unit time and the block rate defines the number of transactions blocked per unit time due

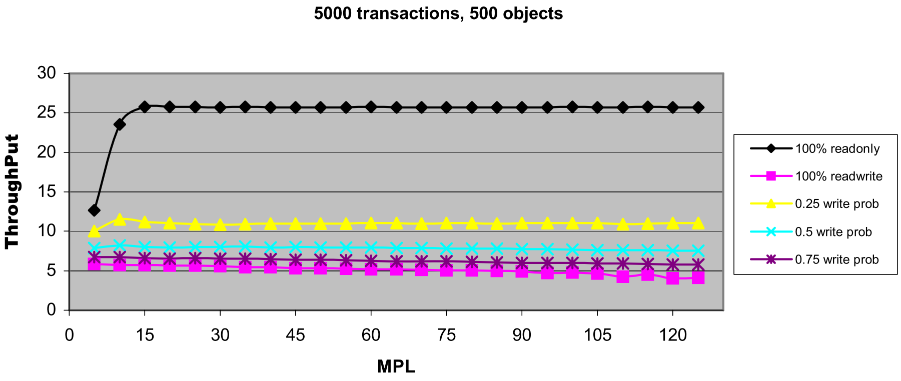


Fig. 8 Throughput vs MPL

to data conflict. We also measured conflict rate; the number of conflicts suffered by concurrent transactions per unit time.

Performance metric 1: throughput vs MPL In this experiment (Fig. 8), we measured the throughput for different MPL with a varying write probability *Write_Prob*. We observe that the throughput remains nearly constant after an initial increase and begin to fall with increase in MPL and write-probability. Throughput also falls with increase in write-probability due to resource contention among transactions. The throughput is much higher for write-probability 0 when read-only transactions are present in the system, as versions are made available much before termination of the transactions. On the other hand for the Write-Probability of 1; when all the transactions are read-write, the throughput is the lowest as each transaction reads and writes. These results are expected from the behavior of the MV-T model described earlier.

Performance metric 2: abort rate vs MPL We observe an interesting result here. In Fig. 9 we have calculated the rate of transaction abort with MPL for different write probabilities. There is an increase in abort rate with lower MPL values for different write-probabilities but rate falls as the MPL increases. This is due to the blocking of conflicting transactions based on timestamps rather than aborting them. However, the initial trend shows that the abort-rate increases as the number of blocked transactions are relatively low compared to aborted transactions but the abort-rate gradually decreases with higher write-probabilities as the number of blocked transactions is much higher than the number of aborts. The interesting pattern is that at higher MPL abort-rate starts increasing again with higher write-probabilities. Therefore, the scheme performs efficiently in the sense that the abort-rate decreases with increase in write-probability for medium load and perform otherwise for very low or very high MPL. This shows that the MV-T model is suitable in mobile environment where aborts are very expensive.

Another interesting result is that for higher write-probabilities the abort rate is less compared to the lower write-probabilities, which is different than what has been observed by other blocking transaction models. This happens because of the blocking

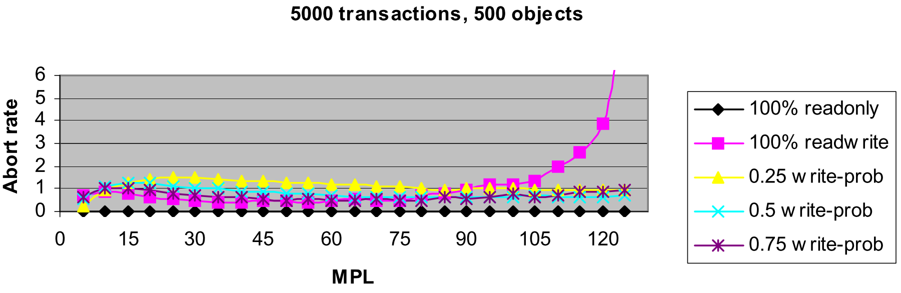


Fig. 9 Abort-rate vs MPL

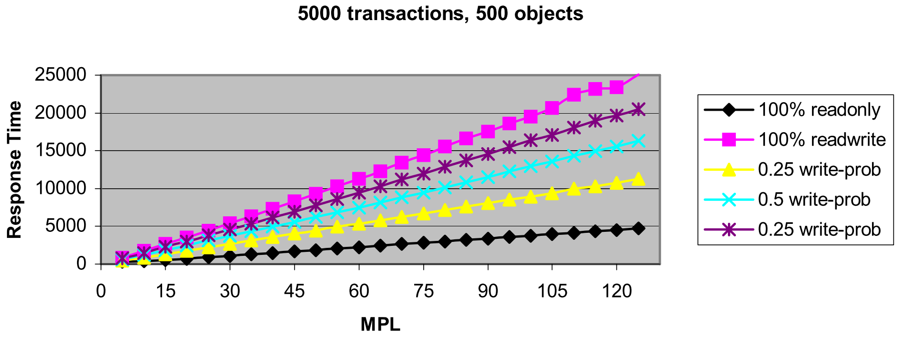


Fig. 10 Response time vs MPL

of the conflicting lock requests present in our model. Note that our MV-T model uses locking as well as timestamps, but the abort-rate for medium load shows a balanced model, neither many aborts as expected in completely optimistic scheme or nor any deadlock based aborts as expected in lock-based schemes.

Performance metric 3: response time vs MPL In this experiment (Fig. 10) we calculated the response time. Graph shows that the response time for a MV-T transaction increases with MPL. It happens because of increasing contention for data objects with the increase in the number of transactions active at any particular time in the system. Another point to note is that the response time for the transaction with write probability 1 is the highest and it is the lowest for the transaction with write probability 0. In case of read-only transactions response time is higher as read-only transactions do not wait for the termination of write-transactions.

Performance metric 4: restart rate & block rate vs MPL In this experiment (Fig. 11), we observed the restart rate and the block rate for different write-probabilities. The graphs on the top portion depict the block-rate and the graphs at the bottom portion depict the restart-rate of the transactions. It is clear that the restart-rate of the transactions is much less compared to the block-rate for any write- probability. The block-rate starts increasing with the increase in MPL and it remains constant for a certain amount of time with MPL between 45 to 60 and then starts to rising

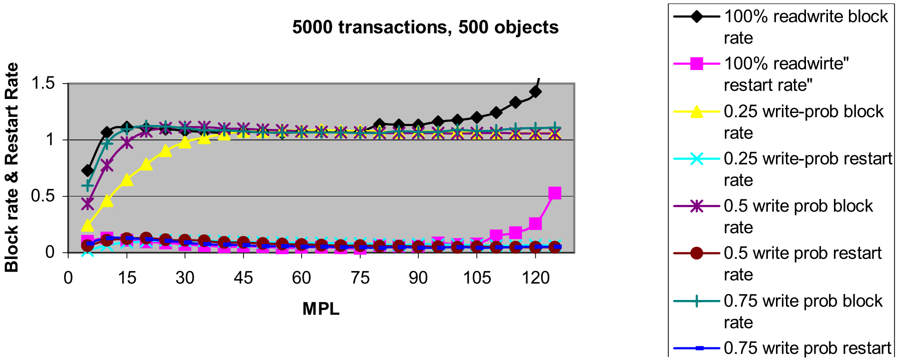


Fig. 11 Block rate & restart rate vs MPL

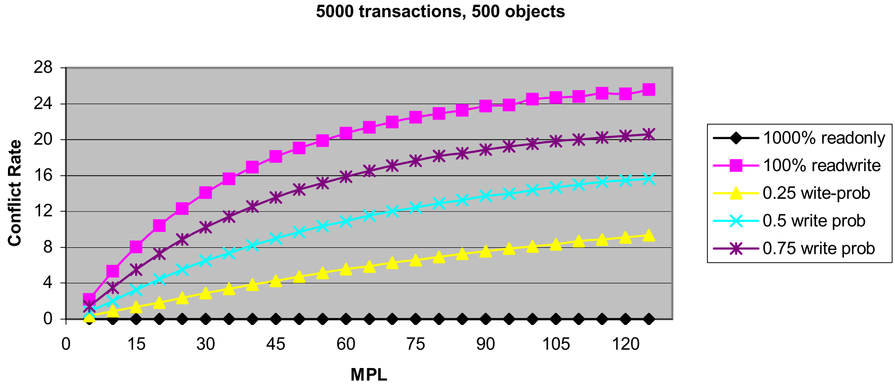


Fig. 12 Conflict rate vs MPL

rapidly again. This gradual rise in the blocking-rate of transactions is dependent on the write-probability, i.e., it occurs first in write probability of 1 then in write probability of 0.75 then in 0.5 and lastly in 0.25. It should also be noted that the rise in block-rate at the start is for much shorter duration for higher write probabilities of 1, 0.75 and 0.5 as compared to lower write-probability of 0.25. The restart-rate performs in the same fashion as the block- rate. This experiment shows that the model achieves a consistent restart rate/block rate for MPL from 45 to 60 and after that limit, the resource contention has increased again, which results in higher restart/block rates. This is also confirmed by the response time as shown earlier where the response time increases more gradually after MPL of 45.

Performance metric 5: conflict rate vs MPL In this experiment (Fig. 12), we observed the rate of conflict with the increasing MPL for different write-probabilities. There is an increase in the number of conflicts with the increase in MPL and the number of conflicts for a particular MPL also increases with the increasing write-probability. Therefore, the highest conflict graph represents the conflict rate for write

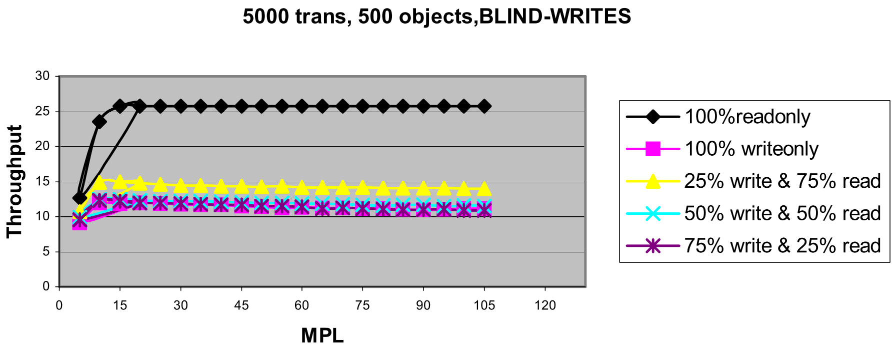


Fig. 13 Throughput vs MPL

probability 1 and the lowest conflict graph represents the conflict rate for the write-probability 0. The interesting observation is that though the conflict–rate increases with gradual increase in write-probability but abort-rate decreases. This behavior is what a mobile system desires due to expensive cost of aborts.

8.2 Performance study with blind writes

In this section, we analyze the different performance metrics evaluated in the previous section in a blind write environment. In the previous case, depending on the write probability, a read-write transaction would perform a read on data object before it writes a version for that data object. In this case, the transaction performs only blind writes, i.e., it does not read the data object but simply writes a new version for the data object. In mobile computing, stock indices, currency rates, etc. can be updated without necessarily reading their earlier values and hence, are considered as blind writes. Here we obtain performance metrics for our model under such an environment. In most of the cases, the graphs behave in a different manner.

Performance metric 1: throughput V/S MPL This graph is very similar to its counterpart without blind writes, where the throughput remains nearly constant with an increasing MPL of more than 25 for a given write probability. The difference is that the graphs for 50%, 75% and 100% write probability are very close and nearly overlaps in this case which is not the same in the *throughput vs MPL* graph for transactions without blind writes. This is because of the absence of reads on data objects before the writes are performed. However, there are read-only transactions mixed with blind write transactions.

Performance metric 2: abort ratio V/S MPL In this experiment (Fig. 14), we calculated the rate of transaction abort with an increasing MPL for different write probabilities in the similar manner as we did for the transactions without blind writes. For any particular write-probability the rate of abort first increases with the increasing MPL but then after certain MPL the rate of abort falls with the increase in MPL and after sometime it starts to rise again with increase in MPL. This later increase

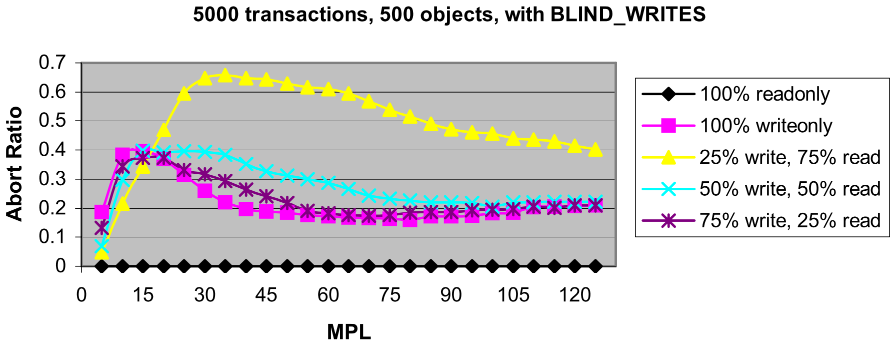


Fig. 14 Abort-ratio vs MPL

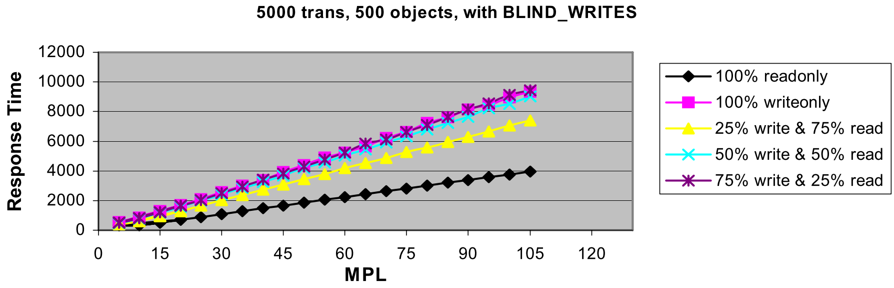


Fig. 15 Response time vs MPL

in the rate of abort with the increase in MPL depends on the write-probability. For a higher write probability this increase appears earlier than compared to the lower write-probabilities That is, the graph starts to rise again for 100% write-only sooner than the rise in graph for 75% write & 25% read case, and so on. In any other locking protocol, the rate of aborts for higher write-probability must be greater than the rate of aborts for the lower write-probability. That is, the graph for 25% write-probability is higher than the graphs for 50%, 75% and 100% write-probability. This is because of the blocking nature of the conflicting write lock requests present in our protocol. Instead of aborting the conflicting write lock requests, we block those with higher timestamp order and thus, resulting in lesser abort rate as compared to the other models.

Another observation to be made here is that with higher write probability the faster is the abort-rate increase for increasing MPL. That is, the graph for 100% write-only begins to fall at MPL of 10, the graph for 75% write & 25% read begin to fall at MPL of 15 and the graph for 25% write & 75% read begins to fall at MPL of 30. This is because of blind writes presence in this case.

Performance metric 3: response time v/s MPL In this experiment (Fig. 15), we have calculated the average time a transaction spends in the system, called the response time. The graphs here behave in the same manner as their counterpart in the earlier

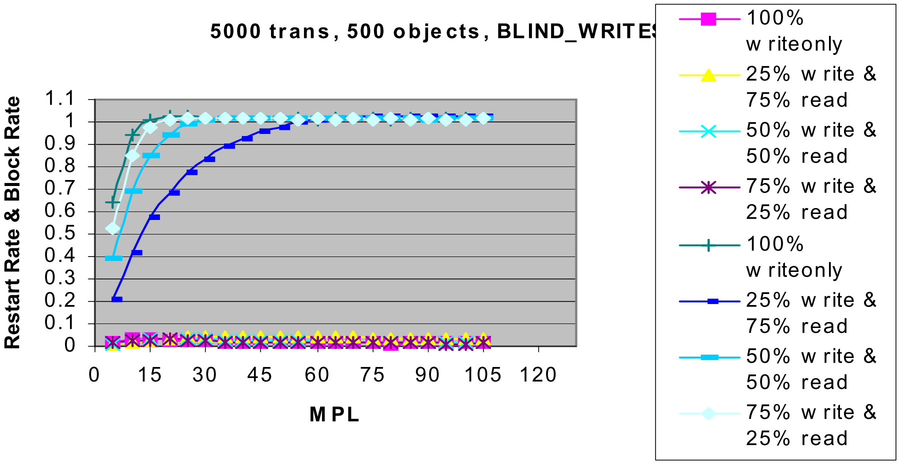


Fig. 16 Restart rate & block rate vs MPL

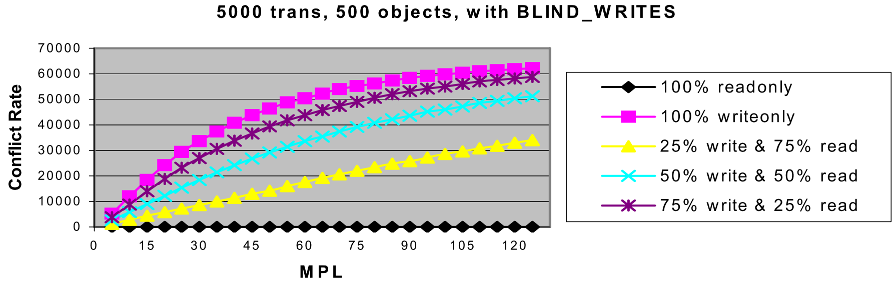


Fig. 17 Conflict rate vs MPL

case, that is, the transactions with a read before every write on a data object. The difference is that the response time is less for any particular MPL and write probability, in this case, because of the absence of read on data objects before writes.

Performance metric 4: restart rate & block rate vs MPL In Fig. 16, we have plotted the block rate and the restart rate for the different write probabilities. The graphs in the top portion represent the block rate and the graphs in the bottom portion represent the restart rate. The graph for a particular write probability behaves in a similar manner as compared to its counterparts in the earlier case where read-write transactions do not have blind writes. The difference is that the block rate graphs tend to remain constant with increase in MPL whereas, in the earlier case, those graphs begin to rise after certain MPL depending on the write-probability. The restart rate of transactions is less compared to the block rate for any write probability, because of the blocking nature of the write lock request present in our model.

Performance metric 5: conflict rate V/S MPL In this experiment (Fig. 17), we have plotted the rate of conflict with increasing MPL for different write probabilities. The

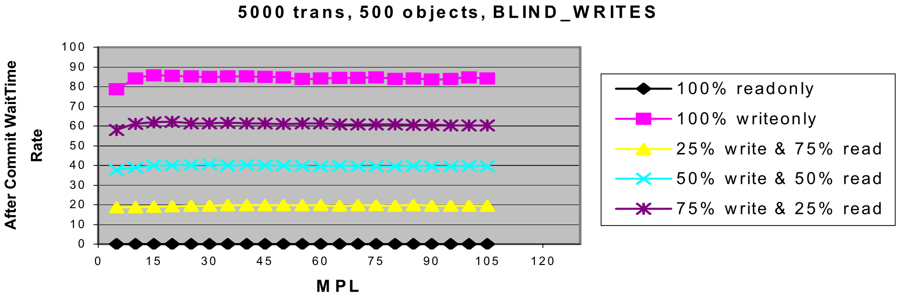


Fig. 18 After commit wait time vs MPL

graphs in the above diagram behave in a similar manner as compared to their counterparts in the earlier case where the write on data object is performed only after read.

Performance metric 6: AfterCommitWaitTime V/S MPL In this experiment (Fig. 18), we have plotted the rate of AfterCommitWaitTime with the increasing MPL for the different write probabilities. *AfterCommitWaitTime* is the average time a transaction has to wait at DBS after its commit before it is actually terminated by DBS. It can be clearly seen that the AfterCommitWaitTime for a particular write probability is nearly constant with varying MPL. That is, even with an increase in MPL, which means an increase in data contention and increase in the number of transactions that are active at DBS, the average time a transaction has to wait after its commit before it is actually terminated, is the same. This behavior is definitely to an advantage of the locking protocol adapted in our model.

8.3 Comparison with speculative transaction model

In this section, we compared our MV-T model with a Speculative Model for Transaction Processing (SMTP) proposed in [38, 39]. We use throughput and response time for the comparison. Note that the SMTP model assumes no aborts as the conflicting transactions are executed in parallel on multiple versions, and each transaction commit on one value and the other one is discarded. That is, a waiting transaction speculatively carries out alternative executions on both before and after images and they retain the appropriate execution outcome based on the termination decision of preceding transactions before its own commit. We selected this model as it uses versions to execute transactions in a mobile environment. Moreover, the SMTP model outperforms traditional 2PL [15], and WDL [16] and therefore, these basic reference models have not been compared again here. In our experiments, we consider only two versions in both the models and keep the same environment as in earlier experiments. Note that this is best case scenario in both the models. Also, in this experiment, each transaction reads and writes a data object (100% read-write) which is the condition used in the SMTP model. Also, this is the worst-case comparison, as the resource contention will be much higher under this condition which makes it useful for the purpose of comparing two models.

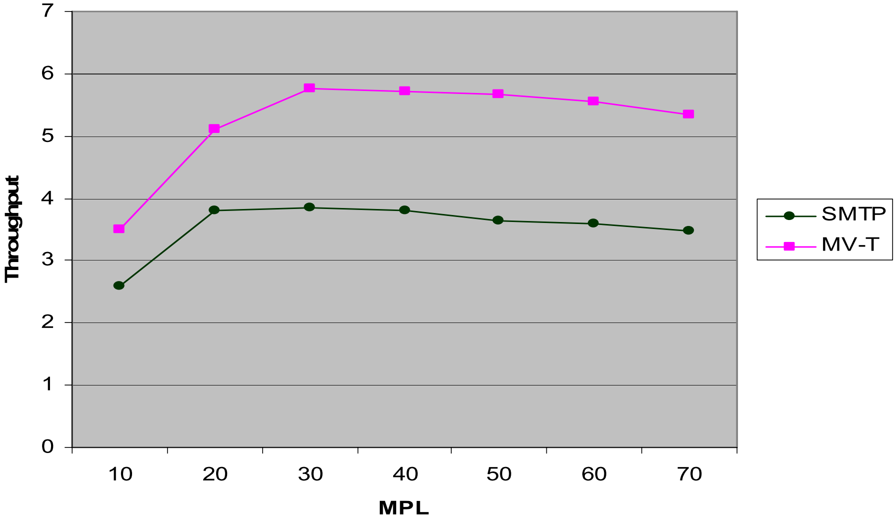


Fig. 19 Throughput: SMTP V/S MV-T

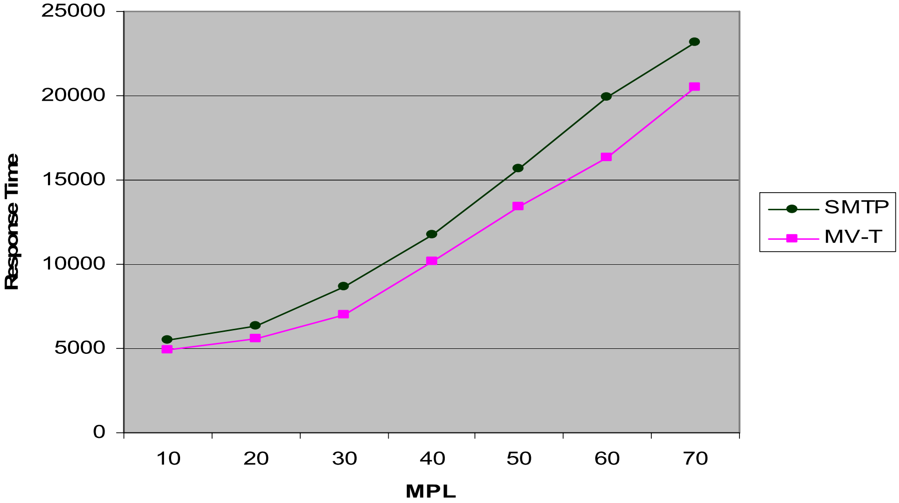


Fig. 20 Response: SMTP V/S MV-T

It is clear from the performance model, MV-T model performs better in terms of throughput as the SMTP takes longer in execution as it compute the values using both the before- and after-versions and the next transaction waits for the completion of earlier transaction and then again waits for the commit of prior transactions. Thus, a transaction has to wait longer for both read and write operations in SMTP. In MV-T model, a transaction upgrades a write-lock to verified-lock to make data values available for others much before it terminates. In addition, in MV-T model, a transaction makes available data values at the time of commit (before final termination), whereas in case of SMTP, they are only available after a transaction is terminated. The advantage of separating a commit state from the termination state is visible from this particular experiment. In addition to the above arguments which also hold true in arguing for increase in the response time in case of SMTP than MV-T model, our model voids delays due to deadlocks by using timestamps. In terms of gradual increase, both models experience resource contention at almost same MPL of about 30 onwards. At a lower value of MPL, the difference in the two models is not that significant as the lock waiting time and the difference between the commit and termination time in MV-T model is comparable to SMTP model's speculative execution of transactions. However, once the load on the system increases, the SMTP model suffers due to resource contention, long-duration, possible deadlocks, and waiting to commit of the previous transactions.

Note that in the presence of read-only transactions, SMTP model's performance will deteriorate as read-only transactions will read both the versions and wait for the previous write transaction to commit on one of the versions. This will not only increase the response time and decrease the throughput, but will also consume extra resources. In addition, SMTP assumes unlimited resources not an appropriate assumption in case of mobile computing. Moreover, every time a transaction is executed, the system accepts only the outcome of the execution using either before- or after-images, and discard the other one and thus, SMTP wastes resources such as storage, battery power and CPU usage. One can argue that in SMTP model discarding a partial result of a transaction (though not considered as abort in [38, 39]) is equivalent to one sub-transaction abort per transaction execution (with only two versions) which is very expensive in case of a wireless mobile environment.

9 Conclusions and future work

In this paper, we presented a formal transaction-processing model that handles two versions: committed and terminated and discussed the locking algorithms to control the concurrent executions of transactions under multiversion. Read-only transactions always return the current value either committed or terminated. We have presented the performance evaluation of MV-T model which increases availability in mobile computing environment. The model presented is deadlock free and reduces abort-rate with increase in write-probability for the moderate load. The blocked transactions also have low restart-rate. Thus, the model provides very efficient transaction execution in a mobile environment with high throughput with low abort-rate. For future work, we are simulating the model in the environment of multiple versions rather than restricting to only two versions. Also, we are considering another variation of the model where we block every conflicting transaction and abort transactions that involve in deadlocks. Another interesting issue we are exploring is to introduce real-time constraints in the model, and transactions which have hard-deadlines, can always read the latest version of the object as reads are never aborted in our model. We are exploring to adapt the model for broadcast transaction processing [11, 25, 34, 40, 41] and in M-P2P domain.

Acknowledgement We are very thankful to the coauthor P.K. Reddy of SPTM model for providing the simulator and fruitful discussions during the period of this research.

References

1. Agrawal, D., Abbadi, A.E.: Constrained shared locks for increasing concurrency in databases. J. Comput. Syst. Sci. **51**, 53–63 (1995)
2. Agrawal, D., Krishnamurthy, V.: Using multiversion data for non-interfering execution of write-only transactions. In: Proceeding of the ACM SIGMOD Conference, pp. 98–107 (1991)
3. Agrawal, D., Sengupta, S.: Modular synchronization in multiversion databases: version control and concurrency control. In: ACM Proceedings of SIGMOD, New York, May 1989, pp. 408–417 (1989)
4. Barbara, D.: Mobile computing and databases—a survey. IEEE Trans. Knowl. Data Eng. **11**(1), 108–117 (1999)
5. Böse, J.-H., Böttcher, S., Gruenwald, L.: Research issues in mobile transactions. http://drops.dagstuhl.de/opus/volltexte/2005/168/pdf/04441.SWM5.Paper.168.pdf (2005)
6. Bober, P., Carey, M.J.: On mixing queries and transactions via multiversion locking. Technical report, Computer Science Department, University of Wisconsin-Madison, Nov 1991
7. Bernstein, P., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, Reading (1987)
8. Böttcher, S., Gruenwald, L., Obermeier, S.: Reducing sub-transaction aborts and blocking time within atomic commit protocols. In: BNCOD, pp. 59–72 (2006)
9. Barghouti, N., Kaiser, G.: Concurrency control in advanced database applications. ACM Comput. Surv. **23**(3), 269–317 (1991)
10. Chrysanthis, P.K.: Transaction processing in a mobile computing environment. In: Proceedings of IEEE Workshop on Advances in Parallel and Distributed Systems, October 1993, pp. 77–82 (1993)
11. Chung, I., et al.: Taxonomy of data management via broadcasting in a mobile computing environment. In: Mobile Computing: Implementing Pervasive Information and Communication Technologies. Kluwer Academic, Dordrecht (2002)
12. Chan, A., Fox, S., Lin, W., Nori, A., Ries, D.: The implementation of an integrated concurrency control and recovery scheme. In: ACM Proceedings of SIGMOD, pp. 184–191. ACM, New York (1982)
13. Dirckze, R., Gruenwald, L.: A pre-serialization transaction management technique for mobile multidatabases. MONET **5**(4), 311–321 (2000)
14. Eich, M.H., Helal, A.: A mobile transaction model that captures both data and movement behavior. In: ACM/Baltzer Journal on Special Topics on Mobile Networks and Applications (1997)
15. Eswaran, K.R., Gray, J.N., Lorie, R.A., Traiger, I.L.: The notions of consistency and predicates locks in a database system. Commun. ACM **19**(11), 624–633 (1976)
16. Franaszek, P.A., Robinson, J.T., Thomasin, A.: Concurrency control for high contention environments. ACM Trans. Database Syst. **17**(2), 304–345 (1992)
17. Goel, S., Bhargava, B., Madria, S.: An adaptable constrained locking protocol for high data contention environments. In: Proceedings of IEEE for 6th International Conference on Database Systems for Advanced Applications (DASFAA'99), Taiwan, April 1999
18. Hwang, S.-Y.: On optimistic methods for mobile transactions. J. Inf. Sci. Eng. **16**, 535–554 (2003)
19. Kumar, V.: Performance of Concurrency Control Mechanisms for Centralized Database Systems. Prentice Hall, New York (1996)
20. Kumar, V., Prabhu, N., Dunham, M., Seydim, Y.A.: TCOT—a timeout-based mobile transaction commitment protocol. IEEE Trans. Comput. **51**(10), 1212–1218 (2002)
21. Kisler, J., Satyanarayanan, M.: Disconnected operation in the coda file system, ACM Trans. Comput. Syst. 10(1) (1992)
22. Kataoka, R., Satoh, T., Inoue, U.: A multiversion concurrency control algorithm for concurrent execution of partial update and bulk retrieval transactions. In: Proceedings 10th International Phoenix Conference on Computers and Communications, pp. 130–136. IEEE Computer Society Press, New Jersey (1991)
23. Kuruppillai, R., Dontamsetti, M., Cosentino, F.J.: Wireless PCS. McGraw-Hill, New York (1997)
24. Lam, K.-Y., Li, G., Kuo, T.-W.: A multi-version data model for executing real-time transactions in a mobile environment. In: Proceedings of MobiDE, pp. 90–97 (2001)
25. Lee, V., Lam, K., Son, S., Chan, E.: On transaction processing with partial validation and timestamp ordering in mobile broadcast environments. IEEE Trans. Comput. **51**(10), 1196–1211 (2002)
26. Lu, Q., Satyanaraynan, M.: Improving data consistency in mobile computing using isolation-only transactions. In: Proceedings of the fifth Workshop on Hot Topics in Operating Systems, Washington, May 1995

27. Madria, S.K., Bharat, B.: A transaction model to improve availability in mobile computing environment. Distrib. Parallel Database Syst. J. **10**(2), 127–160 (2001)
28. Madria, S., Baseer, M., Bhowmick, S.: Multi-version transaction model to improve data availability in mobile. In: Proceedings of 10th International Conference on Co-operative Information Systems (COOPIS'02). Lecture Notes in Computer Science, vol. 2519, pp. 322–338. Springer, Berlin (0000)
29. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4), 631–653 (1979)
30. Sanjay, K.M., Mukesh, K.M., Sourav, S.B., Bharat, K.B.: Mobile data and transaction management. Inf. Sci. **141**(3–4), 279–309 (2002)
31. Pitoura, E., Bhargava, B.: Building information systems for mobile environments. In: Proceedings of 3rd International Conference on Information and Knowledge Management, pp. 371–378 (1994)
32. Pitoura, E., Bhargava, B.: Maintaining consistency of data in mobile computing environments. In: Proceedings of 15th International Conference on Distributed Computing Systems, June 1995; Extended version has appeared in IEEE TKDE (2000)
33. Pathak, S., Badrinath, B.R.: Multiversion reconciliation for mobile databases. In: Proceedings of IEEE International Conference on Data Engineering (ICDE), pp. 582–589 (1999)
34. Pitoura, E., Chrysanthis, P.: Multiversion data broadcast. IEEE Trans. Comput. **51**(10), 1224–1230 (2002)
35. Pitoura, E., Samaras, G.: Data Management for Mobile Computing. Kluwer Academic, Dordrecht (1998)
36. Pu, C., Kaiser, G.: Hutchinson, X.: Split-transactions for open-ended activities. In: Proceedings of the 14th International Conference on Very Large Databases (VLDB) (1988)
37. Ramamritham, K., Chrysanthis, P.K.: A taxonomy of correctness criterion in database applications. J. Very Large Databases **4**(1), 85–97 (1996)
38. Reddy, P.K., Kitsuregawa, M.: Speculative lock management to increase concurrency in mobile environments. In: Proceedings of First International Conference (MDA'99), Hong Kong, China, 16–17 December 1999, pp. 82–96 (1999)
39. Reddy, P.K., Masaru, K.: Speculative locking protocols to improve performance for distributed database system. IEEE Trans. Knowledge Data Eng. **16**(2), 154–169 (2004)
40. Shigiltchoff, O., Chrysanthis, P., Pitoura, E.: Multiversion data broadcast organizations. In: 6th East European Conference on Advances in Databases and Information Systems, ADBIS 2002, pp. 135–148 (2002)
41. Shanmugasundaram, J., Nithrakashyap, A., Sivasankaran, R.M., Ramamritham, K.: Efficient concurrency control for broadcast environments. In: SIGMOD Conference, pp. 85–96 (1999)
42. Serrano-Alvarado, P., Roncancio, C., Adiba, M.: A survey of mobile transactions. Distrib. Parallel Databases **16**(2), 193–230 (2004)
43. http://media.wiley.com/product_data/excerpt/28/04714190/0471419028.pdf
44. Weihl, W.E.: Distributed version management for read-only actions. IEEE Trans. Softw. Eng. **13**(1), 55–64 (1987)
45. Walborn, G.D., Chrysanthis, P.K.: Supporting semantics-based transaction processing in mobile database applications. In: Proceedings of 14th IEEE Symposium on Reliable Distributed Systems, September 1995, pp. 31–40 (1995)