A Lazy Commit Protocol for Mobile Transactions

Sharvanath Pathak
Dept. of Computer Science and Engineering
Indian Institute of Technology Guwahati
Guwahati, India
s.pathak@iitg.ernet.in

Gautam Barua
Dept. of Computer Science and Engineering
Indian Institute of Technology Guwahati
Guwahati, India
gb@iitg.ernet.in

***Abstract*—With an increasing use of mobile and hand-held computing devices, there is a need for new algorithms for data and transaction management in mobile environments. Devices are becoming more and more computationally capable and in many cases power is no more a critical issue: for example, laptops, which can be charged time to time. The reliability of communication links is also not of concern. The bottleneck in many of these situations is turning out to be the communication bandwidth. In this work we present a mobile transaction management protocol, which employs a lazy commit strategy to minimize the bandwidth utilization and the frequency of communication. We defer the commit and lock release until some other device requests a conflicting lock or the user explicitly asks the system to commit the changes. This reduces the communication frequency and also the bandwidth usage. Simulation results show that, in terms of bandwidth usage, our protocol performs strictly better than an existing optimistic protocol for mobile transactions.**

Keywords—Distributed Systems; Transactions; Mobile Computing; Nomadic Computing

I. INTRODUCTION

The standard way of accessing a shared database is to use a browser as an interface to a centralized database. This scheme works very well as long as the network connection is reliable and has sufficient bandwidth. But when a client is using a mobile device the client typically uses a wireless connection and such a connection imposes limitations on the available bandwidth. Further, the amount of bandwidth available is variable as congestion increases or decreases. In such a scenario, there is a need to reduce the amount of communication between a client and the central database. This can only be done by doing more at the client, and in particular, by buffering changes and thereby delaying transmission to the central site. As soon as we do this, the support for performing transactions becomes a challenge. This is the problem we have tried to explore in our work. We describe the problem of mobile transactions in more detail in section 3.

Normally when mobile devices are used, energy consumption is a major concern due to the limited battery life of mobile devices. This work does not consider energy issues as the applications considered are likely to be using tablets and notebooks as clients and in locations where sources of power are likely to be available. Thus, we are considering clients that are relatively static during periods of interaction, and not clients in rapid movement while accessing a remote database.

The rest of this paper is structured as follows. Section 2 describes the system model for general mobile computation and the system assumptions made by us. Section 3 describes the problem of mobile transactions and some existing approaches to solve it. Section 4 provides a brief review of related work. Section 5 describes our approach in some detail and discusses the reasoning behind some of the design choices made by us. Section 6 corroborates our claims with the help of simulation, and it also compares our protocol to an existing optimistic protocol for mobile transactions. Finally, we list the scope for future work and conclude in section 7.

II. SYTEM MODEL

Our system model is a special case of the general one described in [8]. The system consists of mobile devices or MUs(mobile units) and MSS(mobile support stations). Each MU connects or communicates to the system through its nearest MSS. In our assumptions the MUs are mostly connected to one of the MSS and they are adequately powered to do relatively complex tasks. A cell of an MSS is defined as the range of area within which any mobile device will contact to this MSS. The MSS are connected through a wired network and this backbone network is nothing but a typical example of a distributed system. Some specialized nodes in this distributed system behave as DBSs(database servers).

We consider only one MSS in this paper to simplify the discussion and so the term MSS and server are used interchangeably. In a system with multiple MSS a hand-off protocol is needed when a node moves from the cell of one MSS to another. In case a transaction is in progress when the device migrates, a special protocol is needed to guarantee transactional semantics. Many other works like [10] make this assumption to demonstrate their protocol, specifically the ones where mobility is not the major problem being tackled. In our case the major concern is reducing the communication frequency and bandwidth usage.

III. MOBILE TRANSACTIONS

There are a lot of situations when the devices are not only listeners but also are capable of doing updates to the database. For instance, consider an agent booking orders through a laptop on some remote server, a web check-in being done at the airport through a mobile hand-held device or an insurance agent analyzing a claim at some remote site. All these scenarios need the devices to be capable of doing updates to the database.

An example in [5] discusses insurance claim processing through a mobile device, in order to justify the need for

transactional guarantee and disconnected operation in case of mobile transactions. As suggested in the paper, the disconnections might be of two types: voluntary, to save power (which is not our major concern) or involuntary. According to our system assumptions, disconnections should be rare but we do handle such rare disconnections. A common approach is submitting a command to the server, and code is executed there. This is typically what happens when we do updates by filling HTML forms through mobile browsers. Submitting work to the server is a good idea when the connection is fast and predictable.But we assume that our connections do not give good response time always. Moreover, there are situations when this approach increases the complexity of interactions. Let us consider an agent who shows some market price for my belongings and then sells them from his mobile device. Clearly he needs the two steps to be done atomically and in isolation. It won't be acceptable if the agent tells me some price and in the next step, when he actually sells, the price declines and I get less. Sending isolated commands as is normally done through browsers might not be a feasible solution in such general situations, because we need some user interaction between the two steps. We can handle the problem by using locks at the server and by using cookies, but all of us have experienced the frustration of doing an online booking through an erratic internet connection. Time-outs and restarts take place frequently. So a solution where some of or all of the data can be kept and handled locally, even if temporarily, has merit.

IV. RELATED WORK

The following are some of the existing approaches in the area of mobile transactions:

- An optimistic approach for concurrency control is described in [12]. The protocol describes how caching and an optimistic approach can be used to achieve serializable schedules. Invalidation reports are periodically broadcast; a device does the transactions locally; if it doesn't have required data in its cache it requests the MSS. If the data is newer (belongs to the next invalidation than the other in this transaction) or on submission to the MSS it conflicts with other concurrent data items, the transaction is aborted and redone. The benefits of this approach are limited bandwidth usage by limiting the period of invalidation report and more responsiveness, as there is no need of locking. The cases of disconnections are also very naturally handled. Perhaps the single biggest disadvantage of this approach is, aborting and redoing a transaction is very costly in mobile environments.
- Pessimistic approaches which lock the data items and locally commit the data. Our approach is based on these lines. These approaches are generally criticized because of the blocking in case of a disconnection. Our algorithm minimizes this blocking and at the same time argues that this approach with a slightly different idea can be used in many scenarios to minimize bandwidth usage.
- The scheme in [2] considers a scenario where there are both fixed and mobile clients. Due to variability in bandwidth in mobile clients, locks may be held for relatively longer periods by such clients. It proposes an adaptive 2PL scheme that lowers the priority of mobile clients in case there are competing fixed clients. While the paper does consider bandwidth constraints in mobile clients, the solution is not applicable to an all-mobile client system.
- Clustering is used in [10,7] to merge commits and provide better responsiveness, where reads and writes are of two types: weak and strict. The weak read and writes are local to a cluster, and each cluster maintains a weak copy of the data and makes writes immediately visible. The strict reads are to the globally consistent state for all the clusters. The weak writes are merged with the strong writes if there are no conflicts. In case of conflicts the operations might have to be roll-backed and redone. In this case the propagation of weak writes can be combined or even deferred if some amount of inconsistency is acceptable, thereby saving the bandwidth utilization. This system is useful when weak reads and writes can be allowed by the application.
- Kangaro [5] transactions answer the issue of migration of MTs from the cell of one MSS to another, while a transaction is in progress. The system uses split transactions [11] to split a transaction at points where it is possible and completes rest of the transaction at the next MSS. It does not handle the issue of reducing communication between a client and the server.
- Cedar [13] was developed at CMU. It is an optimistic approach which allows clients to store a substantial part of the database and which includes mechanisms to synchronize the client and server databases. It uses Rabin fingerprinting at row boundaries to create checksums to detect changes. It does not address the issues of fine granularity locking that we consider. Their scheme is not going to be feasible for large databases.
- Gray et al. [6] compare several strategies for transaction propagation and conclude that the lazy-commit approach is the one that scales well and fits into mobile environments.
- Mahajan et al. [9] study a similar problem to ours, but they propose grouping of client updates and so they consider how to send the update log cheaply to the server by grouping updates from a set of clients.
- Goldrush [4] discusses a lazy commit strategy, wherein the devices might go off-line in the middle of a transaction and commit later when they become online.

V. THE LAZY COMMIT PROTOCOL

A. Our Approach

As argued in the previous sections, communication is emerging as the bottleneck for mobile environments in today's scenario. Thus, we have optimized our design to reduce bandwidth usage. We also assume that disconnections in the system are rare. Our approach is to allow the transactions to

operate locally for long durations, and to acquire global locks on the corresponding data items to handle concurrency. This reduces communication frequency, as we do not need to ask for locks, which are already held. The locks are released lazily, which happens when another transaction needs a conflicting lock. Before the locks are released the local commits are made permanent by sending the final commit logs (which is a collective effect of several local transactions) to the DBS (database server).

B. The protocol

In the following sections we provide a complete description of the protocol. All the mobile clients run a **local transaction manager (LTM)**. The **global transaction manager (GTM)** runs on the server and manages the actual consistent global state, global commit logs and global locks. Note that we do not talk about multiple MSS because the protocol is not designed to answer mobility issues. It will of course be worthwhile to extend the protocol to handle mobility for a practical implementation.

LTM (local transaction manager): The transactions are executed locally at the mobile devices and the LTM manages all the local DBMS(database management software) issues. In addition to these, a LTM also coordinates the interactions with the GTM and manages the necessary metadata. We have assumed that 2PL is used for concurrency control, which is the dominant protocol in commercial databases.

Following are the data structures that a LTM manages (other than what is needed for local database management):

- A list of all the global locks held, together with the information about the last transaction(s) that have held these locks (last is defined by the serialization order: if two transactions which have held the locks are not serially ordered both are stored) and have locally completed.
- It maintains a reverse serialization graph for all the transactions, which have not been globally committed. The graph is a DAG (directed acyclic graph) and is defined as follows:
    - A node for each local transaction, which has not globally committed.
    - An edge from T_i to T_j, if an action of T_j precedes and conflicts with one of T_i's actions.
- The commit id for the last global transaction it committed. These ids are globally unique, so they can be maintained in the form: Client-id + commit id., where commit id is a monotonically increasing sequence number.

GTM (global transaction manager): The GTM keeps track of the commit logs, database state and global locks. Following are the data structures that it maintains:

- For each lock: the mobile device which currently holds it and which commit identifier released it last, if it was not preempted.
- For each active global commit operation: a global commit descriptor is stored which contains the following fields:
    - The global commit id.
    - The active commit ids it depends on (what we mean by depends will become clear when we discuss what happens when a commit request comes).

Note that the commit is not an atomic step here, because we grant a lock to a requesting device before the commit operation of the transaction, which last held the lock, gets completed. In order to guarantee the ACID properties we have to maintain these data structures.

The LTM logic:

- Client initiated commit request: This logic is invoked in response to a request to globally commit a transaction T_i, which has been locally completed, is submitted to the LTM.
    - The closure of T_i is calculated. These are all the operations reachable from T_i in the reverse serialization graph. The global commit log, which is a collection of the local commit logs of all these transactions, is calculated.
    - The LTM sends the commit log with the next global transaction ID for this device. The log might be large so it is sent in chunks and with each chunk the information of locks that can be released after the log is written to the database, is also sent. We call these chunks and lock release information, the commit-log pages. An implementation issue here is, which global locks should be released on client initiated commit: usually among the locks related to this commit, only those locks should not be released which are very often accessed by this client.
- Local lock acquire request: This logic is invoked when a request for a read or write lock is submitted to the LTM by a local transaction.
    - The LTM checks whether the lock is already present with it.
    - If yes it uses the standard DBMS strategy for lock allocation. Otherwise it sends an acquire/upgrade request to the GTM. After getting the lock it uses the standard DBMS strategy for lock allocation.

- Local lock release request: This logic is invoked when a request to releasea read or write lock is submitted to the LTM by a local transaction.
    - The LTM frees the lock and is ready to allocate it to other local transactions. It does not free this lock globally and keeps it until a global release requestfor the lock arrives or a global commit frees the lock.
- Global lock release request: This logic is invoked when the GTM issuesa request to release a lock to a LTM holding it.
    - If the client does not release the lock in a specified time-out period the GTM will preempt the lock and any active transactions of this device, which hold the lock or are dependent on these lock holders, will have to be rolled back and recovered.
    - Any local transactions, which hold the lock and have not committed are aborted by the LTM. The LTM finds the last transaction T_i, which held this lock and has locally completed. It commits T_i with a procedure similar to the one for client-initiated commit.
    - When the commit is done in response to such a release request, the log page for the requested lock is sent to the GTM at the earliest. If a commit for the requested lock is already in progress, then no action is taken, but again the log page for this lock should be sent at the earliest. This is done so that the GTM can allocate this lock early and the blocking time of the requesting transaction is minimized.

The GTM logic:

- Commit request: This logic is invoked when a commit-log page arrives from an LTM.The GTM checks whether some of these locks have been preempted:
    - If yes, abort message is sent to the client with information on the pre-empted locks.
    - Otherwise, a commit-log descriptor is made for the first commit-log page of a commit id. As described earlier these commit ids are globally unique and are assigned by the clients. It writes the logs and whenever it processes a log page (as defined in the LTM commit request handler) it releases the associated locks. If the write lock is not held it responds with an illegal write message. When a commit is in response to a release request, the requested lock should be released at the earliest in the log pages. When a commit is completed, the id of the transaction must be removed from the "depends" list of any commit descriptors.
- Lock acquire request: This logic is invoked when a write lock-acquire request arrives from an LTM.
    - If any conflicting locks are held by some client(s), they are sent a lock release request. After the time-out period, if all the conflicting locks are released then the lock is assigned directly; otherwise the conflicting locks are pre-empted. If any active commit descriptors hold these pre-empted locks they are aborted.
    - When a lock is granted without an abort, the previous lock holders' commit ids are added to the "depends" list of this commit operation's descriptor, if the commit has not already completed.
- Aborting global commits: This can happen because of a timeout from the client after a commit of that client has begun. It can also happen if an LTM explicitly aborts after the commit starts. Such a message may arrive from an LTM after release of locks because the commit of the transaction holding the locks could not be done. As a third case this can happen for a transaction T_i in response to the abort of a transaction T_j on which T_i depends.
    - Any locks released by the commit descriptor are set to the preempted state.
    - The commit is undone. Any other commit descriptors, which depend on this commit operation, are also undone.
    - Any connected devices are informed so that they can redo these transactions early.

VI. OPTIMIZATIONS

In this section we discuss some simple optimizations, which can be applied for an efficient implementation of the protocol.

Caching & Replication: As mentioned, a standard way to increase responsiveness, in the face of low bandwidth wireless connections, is caching. Our caching mechanism uses invalidation messages similar to the one described in [3]. We maintain a version number for each data entry.

- Replication: The devices store a lot of relevant data locally. The Server broadcasts an invalidation message on writes, together with updated values and a version number, periodically. When a client tries to acquire a lock, it gets the version number for that row or table or index piggybacked with the response. If the version number of the stored copy is not the same as that of this one it has to fetch the latest copy.
- Caching: We don't need invalidation reports at all, because if the entry is locked locally there is no chance of updates and if the lock is already released the version number will be piggybacked with the lock response.

Client Initiated Commit: As the protocol states, transactions remain locally completed for long durations and are globally committed on a lock request. The actual value of this duration is an implementation issue. So even if there are no lock requests a locally completed transaction might be committed globally if the held entries are not being locally used for long durations. This will avoid unnecessary locking of entries. It will also improve the reliability of the system. Clients may also need to ensure that their changes have been pushed to the server at certain times. As an optimization, this client initiated commit should be done when the connectivity is good, as there is an inherent variability of connectivity in a mobile environment.

Clustering: The settings for which we proposed the protocol was primarily where the clients access a disjoint set of pages and there is a fair amount of overlap in their access, so we keep them locked for longer durations. We can employ clustering to ensure that this overlap happens among the nodes of a cluster. The cluster head holds the locks and releases them lazily. If there are many nodes in a cluster the probability of overlap and no contention increases, whereas the communication overhead for lock acquire is low within a cluster.

VII. PERFORMANCE EVALUATION

In this section we study the performance of our protocol using simulation and analytical evaluation. We found that our protocol performs reasonably well as compared to an existing optimistic protocol [12] and strictly better than the naive 2PL version, under the given system parameters and assumptions.

A. The Evaluation Model

We simulated the protocols under several relevant assumptions. As we have already discussed, we make the assumption of only one MSS, because the question we want to answer is bandwidth usage and not mobility. We have simulated the system on an Intel i7 machine with 4GB memory, running Mach OS X 10.6.6. The entire simulation is done in C++, which compiles using g++ version i686- apple-darwin10-g++-4.2.1 and no network simulator is used. We haven't used any network simulator primarily because we aren't interested in the network related events like mobility, disconnections etc. If at all, we might be interested in disconnections to show our protocol handles them, but this is only for correctness verification and not for performance comparison. The correctness appears to be rather trivial in our protocol. Before we move on, let us recall the system model we are motivated by: clients operate with very little contention with other clients, and there is a fair degree of overlap of operations among the transactions of a particular client.

B. Evaluated Protocols

- The naive 2PL protocol: This is the naive 2-phase locking based protocol. It suffers from several limitations, like communication overhead, unnecessary blocking, etc. This is used only to provide a reference point. For the given evaluation parameters it is easy to see that the total size of logs transmitted will be C□T □WS and the number of messages exchanged will be 2□C□T. For each transaction we have two messages, one for acquiring all the locks and one for submitting the log.

- The optimistic protocol: This is the optimistic protocol discussed in [12]. We have simulated it using threads as discussed earlier. We did not implement caching, but that doesn't make any difference, as the comparison is done with a non-caching based version of our protocol. The caching mechanisms for this protocol is very similar to ours, hence it is justified to use the respective non-caching based implementations for performance comparison.

- The lazy commit protocol: Here we analyze the protocol presented in this paper. We have done the exact implementation of the protocol described in section 5.2. As described earlier we do not implement caching. We have not implemented the other optimizations as well, which are discussed in section 5.3.

C. Variation with P0, the probability of no contention.

Here we describe the observations we got by varying the parameter P0. As expected the size of logs transmitted as well as messages sent should decrease by increasing this parameter, because the contention reduces. The size of the log in our protocol reduces even more rapidly because the overlap among the trans- action of same client also increases with increase in P0 as shown in figure 1. Note that the size of database we use in our simulation might appear to be small, but as suggested in [1] this is a common practice in databases to get the results in reasonable time. □Figure 1 shows the performance, with the evaluation parameters: C = 5, T = 10,WS = 10,DB = 660.

An important point to notice was, in some cases we get total messages sent for a client to be even less than T. This can be attributed to the cases where all the locks needed are locally available and there are no extra messages for some transactions. Therefore, if the nature of application very closely obeys our assumptions, we might even outperform the existing protocols in terms of both the message complexity as well as the size of logs transmitted. It is important to note that the total size of logs transmitted in our case is better than the 2PL even with reasonable contention.

D. Discussion

As the experiments demonstrate, we perform better than the existing commit protocols for mobile transactions in terms of the total size of logs transmitted. In terms of messages sent we perform worse than the optimistic protocol, as thenumbers suggest, but this also depends on the invalidation report period, and we have assumed it to be the minimum possible for the optimistic protocol.

An important point to notice was, in some cases we get total messages sent for a client to be even less than T. This can be attributed to the cases where all the locks needed are locally available and there are no extra messages for some transactions. Therefore, if the nature of application very closely obeys our assumptions, we might even outperform the existing protocols in terms of both the message complexity as well as

the size of logs transmitted. It is important to note that the total size of logs transmitted in our case is better than the 2PL even with reasonable contention.

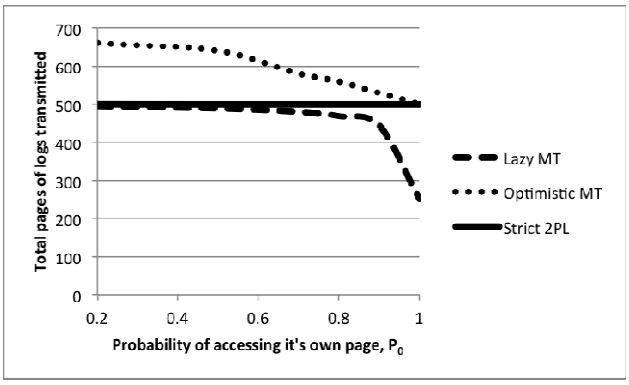

Fig. 1. The variation of messages exchanged and for the three protocols: Lazy, Optimistic and Strict 2PL with the increase in P0 (the probability of a client accessing a page form it's own domain). The measurements are accurate within an error of 0.5 with 95% confidence level.)

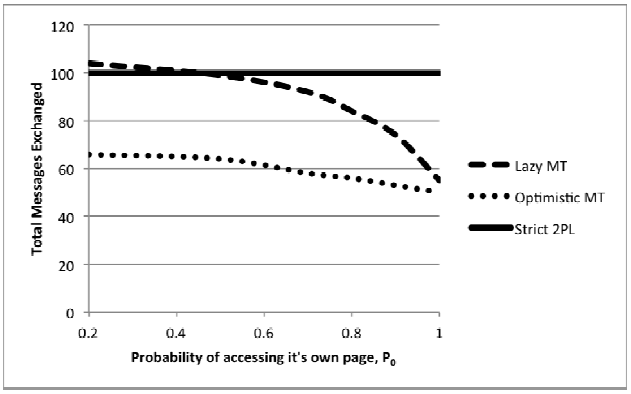

Fig. 2. The variation of size of total logs trasnmitted for the three protocols: Lazy, Optimistic and Strict 2PL with the increase in P0 (the probability of a client accessing a page from it's own domain). The measurements are accurate within an error of 0.5 with 95% confidence level.)

VIII. CONCLUSIONS

In this paper we have described a novel protocol for transaction management in a mobile environment. The protocol optimizes the bandwidth usage and communication frequency. The protocol is based on the observation that in many practical scenarios the commit of transactions can be deferred for a long time as other transactions do not need to operate on the same data very frequently. Deferring such commits can save a lot of communication, as it allows local processing of transactions. We have outlined the working of our protocol, which uses this idea and tries to minimize blocking of transactions, which is the only disadvantage in this case. Our protocol is neither purely optimistic nor purely pessimistic. This hints that a midway protocol is generally better for mobile transactions. Probably, because of the stringent constraints, any of the extremes will not be a good choice.

We have evaluated the protocol against an existing optimistic protocol using simulation. We noticed that our protocol performs quite well in terms of the size of logs transmitted, although the frequency of communication might be little higher in our case. When connectivity is high, bandwidth usage is the more important performance factor. So we can get reasonable gains by deploying this protocol. In fact we noted that if the access patterns we optimized for are closely followed, our protocol could strictly outperform the existing protocols.

REFERENCES

[1] G.R. Agrawal, M.J. Carey and M.Livny,"Concurrency Control Performance Modeling: Alternatives and Implications,". ACM Trans. Database Systems, vol. 12, no. 4, pp. 609–654, November 1987.

[2] A. Alqerem,"Concurrency Control for Mobile Transactions in Presence of Bandwidth Variability," in Proceedings of the 2011 International Conference on Computers and Computing, World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, WI, USA, 2011, pp. 57–60.

[3] D. Barbara and T. Imielinski,"Sleepers and Workaholics: Caching Strategies in Mobile Environments," in Proceedings of the 1994 ACM SIGMOD, International Conference on Management of data,NY, USA, 1994, pp. 1–12.

[4] M.A. Butrico, H. Chang, A. Cocchi, N.H. Cohen, D.G. Shea and S.E. Smith,"Gold Rush: Mobile Transaction Middleware with Java-object Replication," in Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies,Volume 3, Berkeley, CA, USA, 1997, pp. 91–101.

[5] M.H. Dunham, A. Helal and S. Balakrishnan,"A Mobile Transaction Model that Captures Both the Data and Movement Behavior," Mobile Networks and Applications., vol. 2, no. 2, pp. 149–162, October 1997.

[6] J. Gray, P. Helland, P. O'Neil and D.Shasha,"The Dangers of Replication and a Solution," in Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, NY, USA, 1996, pp. 173–182,

[7] S. K. Madria and B. Bhargava,"A Transaction Model for Mobile Computing," in Proceedings of the 1998 International Symposium on Database Engineering & Applications, DC, USA, 1998,pp. 92-102.

[8] S.K. Madria, M. Mohania, S.S. Bhowmick and B. Bhargava,"Mobile Data and Transaction Management," Information Sciences, vol. 141, no. 3-4, pp. 279–309, April 2002.

[9] S. Mahajan, M.J. Donahoo, S.B. Navathe, M.H. Ammar and S. Malik, "Grouping Techniques for Update Propagation in Intermittently Connected Databases," in Proceedings of the Fourteenth International Conference on Data Engineering,DC, USA, 1998, pp. 46-53.

[10] E. Pitoura and B. Bhargava,"Maintaining Consistency of Data in Mobile Distributed Environments," in Proceedings of the 15th International Conference onDistributed Computing Systems,DC, USA, 1995, pp. 403-413.

[11] C. Pu, G.E. Kaiser and N.C. Hutchinson,"Split-Transactions for Open-Ended Activities," in Proceedings of the 14th International Conference on Very Large Data Bases, CA, USA, 1988, pp. 26–37.

[12] San yih Hwang,"On Optimistic Methods for Mobile Transactions," Journal of Information Science and Engineering, vol. 16, pp. 535–554, 2003.

[13] N. Tolia, M. Satyanarayanan and A. Wolbach, "Improving Mobile Database Access over Wide-area Networks without Degrading Consistency," in Proceedings of the 5th International Conference on Mobile Systems, Applications and Services, 2007, pp. 71-84.