MCA-127
A Mechanism for Replicated Data Consistency in Mobile Computing Environments

José Maria Monteiro
University of Fortaleza
Dept. of Computer Science
60811-341 Fortaleza - Brazil
monteiro@unifor.br

Angelo Brayner
University of Fortaleza
Dept. of Computer Science
60811-341 Fortaleza - Brazil
brayner@unifor.br

Sérgio Lifschitz
PUC-Rio
Dept. of Computer Science
22453-900 - Rio de Janeiro - Brazil
sergio@inf.puc-rio.br

Markus Endler
PUC-Rio
Dept. of Computer Science
22453-900 - Rio de Janeiro - Brazil
endler@inf.puc-rio.br

ABSTRACT
The recent advances in portable computer together with the developments in wireless communication technology are allowing users of portable equipment to maintain the network connection while they move about freely, having access to shared resources, services and information. This paradigm is called mobile computing. Mobile computing allows for the development of new and sophisticated database applications. Such applications require the reading of current and consistent data. In order to improve the data availability, increases performance and maximize throughput, data replication is used. However, due to inherent limitations in mobile and other loosely-coupled environments, the concurrency control and replica control mechanisms must be revisited. This paper proposes a new protocol that guarantees the consistency of replicated data in a mobile computing environment, while provide high data availability and ensure an eventual replica convergence towards a strongly consistent state. Our protocol introduces a significant improvement over other solutions by using a consolidated correctness criterion, the serializability, and relaxing the isolation property, beyond, reduces the number of messages exchanged between the hosts and ensure that the transaction operations are executed only one time.

Keywords
Concurrency Control, Data Replication, Mobile Computing

1. INTRODUCTION
The integration of portable computer technology with wireless-communication technology has created a new paradigm in computer science called mobile computing. In a mobile computing environment, network nodes are no longer fixed, that means, they do not have a fixed physical location. In such an environment, mobile users with a portable computer (denoted mobile client or host) may access shared information and resources regardless of where they are located or if they are moving across different physical locations and geographical regions.

Mobile computing technology has made possible the development of new and sophisticated database applications. Electronic commerce applications, such as auctions, road traffic management systems and shared document editing [10], are examples of database applications, which require the support of mobile computing technology. Such applications need to access current and consistent data, even though they concurrently access shared data.

Data replication has been widely used as a strategy to improve the data availability, obtain performance increases and maximize the throughput. The ley idea behind the concept of data replication consists in storing multiple copies of the data items in different servers, distributed in a communication network. Thus, applications can access data items of any replicated server, and are also able to continue their execution, even if some servers fail. However, in a computional environment with data replication all copies of a given data item must represent the same snapshot of the real world. In other words, coherence among replicated data items in several servers should be guaranteed. Data replication has been also used in mobile computing environments. This is because data replication may guarantee data availability despite mobile computing environments restrictions such as low bandwidth, frequent disconnection of the mobile devices, data mobility and large numbers of clients [11].

A replicated database is a distributed database over a wireless network, in which multiples copies of the data items are stored in different servers. Applications accessing data stored in replicated databases need to access consistent copies of replicated data items. Therefore, replicated DBMSs should ensure data consistency. In replicated databases data consistency is guaranteed by applying the following two correctness criterion. First, the replicated DBMS should ensure that the concurrent execution of a set of transactions over a replicated database should be equivalent to a serial execution of the same set of transactions over the same database without replication (one-copy database). This correctness criterion is called one-copy serializability (1SR). Second, the replicated DBMS needs to assure that the state of the diverse copies of a given data item will eventually converge to a unique consistent final state, in spite of the operations executed in the different copies. This correctness criterion is called of eventual consistency. In traditional replicated DBMSs the afore mentioned correctness criterion are guaranteed by a replica control protocol, which can be classified in three groups: Primary-Copy Methods (also called of pessimistic replication), Quorum-Consensus Methods and Available-Copies Methods (also known as optmistic replication) [3]. However, those classes are not efficient to replicated database in mobile computing environments [11]. In order to ensure replicated data consistency in mobile computing environments, several approaches relax one-copy serializability. However, most of these approaches require intensive message exchange [2, 5, 9], or, that transaction operations (stored in log), be re-executed several times. For example, in [13] the servers may need to undo previously executed “tentative write” effects and re-execute them in another order. Thus, a write operation can be executed many times in the same server. Moreover, some proposals allow that the degree of consistency be adapted to several connection levels. In [13] this is obtained by selecting the session guarantees. However, such guarantees don’t have clear semantics from the point of view of data consistency, since, no relation between these guarantees and the standard ANSI/ISO SQL 92 isolation levels [1] is established and the phenomena that are avoided in each session guarantee aren’t specified.

This paper proposes a new mechanism that guarantees the replicated data consistency in mobile computing environments based on one-copy serializability and eventual consistency (the convergence of several replicas to a unique consistent final state). This protocol explores temporal information about the moment where a replicated database object was read or updated. The solution we propose has the main contribution of introducing a significant improvement over other solutions by using a consolidated correctness criterion, the serializability, and relaxing the isolation property to different copies of a given data item. In other words, our approach ensures global serializability, but different transactions may write different copies of a given data item concurrently. Furthermore, the transaction operations are executed only once, the servers don’t need to store a log with all executed operations, and the number of messages exchanged between the servers are reduced. In this way, our protocol tries to ensure a savings in the communication channels costs, in the portable computers limited power capacity and in their scarce memory resources.

The rest of the paper is organized as follows: In section 2 the related works will be described and discussed. Section 3 describes the transactional model that we will use in this paper. In section 4 we describe the proposed protocol for consistency control. Section 5 discusses and analyzes the proposed mechanism. Section 6 concludes this paper and outlines future studies.

2. RELATED WORK

The problem of replicated data consistency in mobile computing environments has been addressed in many different studies, whose main objective is to make high data availability possible. For our purposes it is useful to position existing mechanisms on two axes: centralized versus decentralized, and pessimistic versus optmistic. Centralized systems are characterized as having distinguished entities that play some critical role. In decentralized solutions there isn't distinguishe between entities. Pessimistic systems do not allow data to be updated in multiple disconnected partitions at the same time, ensuring that update conflicts do not occur. Optimistic systems, on the other hand, allow data to be updated across partitions, and attempt to resolve conflicts when disparate versions are later reconnected. We briefly highlights systems in the four categories described by the axes.

Gifford's original weight voting algorithm [6] is an example of a pessimistic, centralized technique. These systems typically provide low data availability. Bayou project [13] is a exemplo of "update anywhere" optimistic, centralized systems. This system allow data to be updated by disconnected clients and conflicts are resolved in a pair-wise fashion during reconnections. In Bayou, updates propagate epidemically and conflicts are resolved by bundling writes with fragments code called conflict resolvers that represent the application's interest and are hard to write. These reconciliation-based protocols are only viable in non-transactional domains such as file systems. Moreover, in the Bayou, each server stores a log containing all the writes that it knows and an operation can be executed several times, which can generate overhead. Buyou don't has support to transactional semantics, it uses a weak consistency approach based on session guarantees. However, such guarantees don't have clear semantics from the point of view of database consistency. Golding [7] proposed an optimistic, decentralized approach in which each server, individually, consolidate an update when it is certain that this update has already been received by all replicated servers. The main limitation of this approach is that the non-availability of any one replicated server makes the commitment process impracticable. The voting based approaches allow the use of a great variety of quorums to decide for the commit of an update. In particular, Deno [5, 8], a optmistic, decentralized approach, uses an epidemic voting protocol in order to support the data replication in a transactional framework for weakly-connect environments. Deno's base protocol provides a weaker consistency model in which writes are not guarantee to serialize with reads. However, Deno requires that one voting round (stage) be completely executed for each update. This can be acceptable when the applications are interested in knowing the commitment process course for each tentative update of a data item, before executing another operation on this same data item. However, in the scenarios considered for this study, users and applications will frequently be interested in access data items which have been updated for non committed transactions. In such situations, the wait for the commit operation, imposed by the Deno voting protocol becomes as unacceptably high as in the primary-commit protocol. In [9] the authors propose a pessimistic, decentralized approach, that is a variant of David Gifford's [6] classic weighted-voting scheme for replicated data. The main limitation of this approach is the low data availability, since this protocol don't use the update anywhere semantics offered by optimistic concurrency mechanisms.

3. TRANSACTION MODEL

A database consists of a collection of objects representing entities of the real world. The set of values of all objects stored in a database at a particular moment in time is called the database state. The real world imposes some restrictions on its entities. Additionally, databases must capture such restrictions, called consistency constraints. If the values of objects of a particular database state satisfy all the consistency constraints, the database state is said to be consistent. The values of these objects can be read or written by the transactions. A transaction is an abstraction which represents a finite sequence of reads and writes operations on database objects. We use the notation $r_i(x)$ and $w_i(x)$ to represent a read and write operation by a transaction T_i on object x. $OP(T_i)$ denotes the set of all operations executed by T_i . We will assume that the execution of a transaction preserves the database consistency, if this transaction runs entirely and in isolation from other transactions.

The concurrent execution of a set of transactions is carried out by the interleaving of the database operations of the various transactions. Some interleavings may produce inconsistent database states. Hence, it is necessary to define when an execution of concurrent transactions is correctly interleaved. Henceforward we will call correctly interleaved execution: correct execution. A global schedule or history indicates the order in which the operations in a set of transactions is executed in relation to the others. Two operations of different transactions conflict (or they are in conflict) if and only if they access the same object of the database and at least one of them is a write operation. The notation $p <_S q$ indicates that operation p was executed before operation q in global schedule S.

Let S be a global schedule on a set $T = \{T_1, T_2, ..., T_n\}$ of transactions. The serialization graph for S, represented by $GS(S)$, is defined as the directed graph $SG(S) = (N, E)$ in which each node in N corresponds to a transaction in T. The set E contains the edges in the form $Ti \rightarrow Tj$, if and only if $Ti, Tj \in N$ and there are two operations $p \in OP(T_i)$, $q \in OP(T_j)$, where p conflicts with q and $p <_S q$. A global schedule S is conflict serializable if and only if the serialization graph for S ($SG(S)$) is acyclic. A schedule S is correct if it is serial or conflict serializable [4].

A replicated mobile database is a database distributed over a wireless network, in which multiple copies of the data items are stored in different servers. The replication unit is called an object. A copy is a copy of an object, stored in a given server. A server can store multiple objects copies, however, sometimes we use the term copy to mean a server (host). The servers that can update their copies are called master servers (master sites), to differentiate from those whose copies (replicated data) are read-only. If the number of master servers is equal to the number of replicated servers, one has a multi-master schema. The proposed mechanism uses the multi-master scheme, that is, read-any/write-any. The servers allow access (read and write) to the replicated data even when they are disconnected.

4. A MOBILE DATA REPLICATION MECHANISM

4.1 Basic Concepts

The proposed mechanism was designed for mobile computing environments composed of a set of replicated database servers and by weakly-connected portable devices. In such environment, a client and a server can coexist in one host (fixed or mobile). The communication channels are unstable and a server is frequently not available or reachable.

Applications do not need to be limited to access only one server, but rather they can interact with any server, that is, they can execute read and write operations on any server with which they can establish a communication link. Thus, a multi-master approach (read-any/write-any scheme) is used.

Copies stored in different servers can present distinct contents. That is, $DB(Re_1, t)$, which is the state of the database in the replicated server Re_1 at instant of time t, is not necessarily equal to $DB(Re_2, t)$ for any two servers Re_1 and Re_2. To reach an eventual consistency in which the servers converge to an identical copy an adaptation in the primary-commit scheme [13] is used. Thus, a server chosen as primary has the responsibility to synchronize and commit the updates, that is, to make them permanent. The committed writes generate new versions of data items that must be propagated to the other servers.

The protocol does not include the notion of disconnected operation (where the client accesses local data, and when the connection is re-established, executes a synchronization procedure) , since several connectivity levels are possible. It is necessary that the communication between each server and the primary server, or service provider, be only occasionally established. Thus, the system deals with intermittent network connectivity. For example, a server can be disconnected from the rest of the system and still be used by the clients.

4.2 Ensuring Mobile Replicated Data Consistency

The proposed approach for ensuring data consistency in mobile replicated databases is based on a similar strategy used by the conventional serialization graph testing protocol [4]: the dynamic monitoring and management of a conflict graph should always be acyclic. In contrast to classic serialization graph testing, this protocol exploits temporal information w.r.t. the moment in which a mobile transaction operation (read or write) is executed on a given database item.

In our approach, we have decided to distribute concurrency control functions among mobile clients, servers and the primary server (consistence service provider). Thus, we assume that the primary server, the other servers (copies) and the clients execute specific functionalities, in order to manage the transaction. In the following, we describe such functionalities.

After the commit of a transaction T_i, the primary server propagates the new values of data items updated by T_i, together with the corresponding timestamps (versions), tor all copies with which it has communication. These timestamps are defined as following:

- Each data item x in $DB(Re_i)$, where $0 < i \leq N$, where N is the number of replicated servers, is associated with a timestamp $C(x)$;
- Each operation $P_i(x)$ (read or write) of a transaction T_i is associated with a timestamp represented by $C(P_i(x))$;
- A timestamp consists of an ordered pair (z, y). The first part (z) is called version, while the second part (y) is called subversion;
- Initially $C(x) = (0, 0)\ \forall x \in BD$, in all copy Re_j, where $0 < j \leq N$;
- The version of $C(x)$, that is, z, is incremented in each commit of a any transaction T_i that executes a write operation on x (new version of x);
- To each commit of a transaction T_i that produces a new version of x, with $C(x) = (v, 0)$, where $v > 0$, this new version, together with its new timestamp, is sent by the primary server for all the replicated servers Re_j (with which it has communication);
- Different strategies can be used for sending the new values of the data items and the new versions, of the primary server (main copy) for the others replicated servers, making possible a means of adaptability, where the context information, such as: signal quality, power level and available memory, can be considered. These strategies will be discussed later;
- The sub-version of $C(x)$, that is, y, is incremented to each write operation of any active (uncommitted) transaction T_i, that it wrote on x, in any replicated server Re_j;

To each read operation $r_i(x) \in T_i$ (that can be executed in any replicated server Re_j), the used replicated server (Re_j) keeps the read data item identifier (id), together with the corresponding timestamp, $C(r_i(x))$, which corresponds to the current value of the read data item timestamp, that is, $C(x)_{Re_j}$, and associates these values to the executed operation $r_i(x)$.

To each write operation $w_i(x) \in T_i$ (that can be executed in any replicated server Re_j), the used replicated server (Re_j)

increases the subversion (y) value of the read data item (x) timestamp value, in its local copy (Re_j). Then, the server (Re_j), keeps the value and the identifier of the updated data item, together with the corresponding timestamp, $C(w_i(x))$, which corresponds to the current value of the timestamp of the read item, x, in its local copy, that is, $C(x)_{Re_j}$, and associates these values to the effected operation ($w_i(x)$).

Periodically, each replicated server (Re_j) must send a package (message) to the primary server containing the read and write operations executed in its local copy (up to the current instance), together with its respective timestamps, as well as its proper identifier (Re_j). The operations already informed do not need to be sent again. This information is received by a scheduler in the primary server, and is used to synchronize the operations of several transactions, generating a correct interleaving, that is, that preserves (according to some correcness criterion) the replicated data consistency.

When a client receives a commit or abort order for a mobile transaction T_i, it must submit this order to any replicated server (Re_j), which will re-send this order to the primary server. In the case of a commit operation it must also send the number of operations of the transaction (T_i). Additionally client C_i can send the message that contains the commit request of a transaction T_i directly to the primary server. The client must await the reply to execute the commit or abort operation. If the timeout runs out and the client does not receive the reply for his commit request, some alternatives can be chosen: abort the transaction T_i or define T_i as a tentative transaction, that is, a concluded transaction whose writes can not be committed. Additionally, if the transaction T_i is a read-only transaction, the client can decide to conclude the transaction, being aware that the transaction may have read inconsistent values. With this purpose, context information can be used as a means of adaptability. For example, if the power level is low and the signal quality is not good, disconnection is imminent. In this case, the client can opt to end his transaction as a try, that is having the knowledge of that its writes can not be committed. In other words, they can be undone, and that its reads may have accessed inconsistent values (temporary). On the other hand, the client can wait a little more and the connection with server Re_j, or with the primary server, will probably be re-established.

The proposed approach to ensure the replicated data consistency is based on the comparison of the timestamps associated with each read or write operation belongs to several transactions that are executed on different replicated servers. The comparison between the two timestamps $C(q_j(x))$ and $C(p_i(x))$ is carried out in the following manner: It assumes, without loss of generality, that $C(q_j(x)) =$ a.b and $C(p_i(x)) =$ c.d. If $a > c$ then $C(q_j(x)) > C(p_i(x))$. Now, if $a = c$, the subversion value must be verified. Thus, if $a = c$ and $b > d$, then, $C(q_j(x)) > C(p_i(x)))$. In the case where $a = c$ and $b = d$, $C(q_j(x)) = C(p_i(x))$.

4.3 Replica Control Protocol

A scheduler implementing the proposed protocol runs as follows. When a scheduler starts its execution an empty graph, called the temporal serialization graph (TSG), is created. Based on the timestamps described on the previous paragraph, the (TSG) is constructed as described below:

Step 1. For each operation $p_i(x) \in OP(T_i)$ received by the scheduler, it checks if exists an operation $q_j(x) \in OP(T_j)$ that conflicts with $p_i(x)$ and that has already been scheduled. In case $q_j(x)$ exists, an edge will be inserted between T_i and T_j, as follows:

```
If $C(q_j(x)) < C(p_i(x))$
  Then the scheduler inserts an edge on the form
    $T_j \rightarrow T_i$
Else
  If $C(q_j(x)) > C(p_i(x))$
    Then the scheduler inserts an edge on the form
      $T_i \rightarrow T_j$
  Else //$C(q_j(x)) = C(p_i(x))$
    If $p_i(x)$ and $q_j(x)$ was execute on the same copy
      If $p_i(x)$ is a write operation and $q_j(x)$ is a read
        An edge on the form $T_i \rightarrow T_j$ will be
        inserted on the graph
      If $p_i(x)$ is a read operation and $q_j(x)$ is a write
        An edge on the form $T_j \rightarrow T_i$ will be
        inserted on the graph
    If $p_i(x)$ and $q_j(x)$ was execute in different copies
      If $p_i(x)$ is a write operation and $q_j(x)$ is a read
        An edge on the form $T_j \rightarrow T_i$ will be
        inserted on the graph
      If $p_i(x)$ is a read operation and $q_j(x)$ is a write
        An edge on the form $T_i \rightarrow T_j$ will be
        inserted on the graph
      If $p_i(x)$ and $q_j(x)$ are write operations
        Then the scheduler inserts an edge on the form
        $T_j \rightarrow T_i$ on the graph
```

Step 2. The scheduler verifies if the new edge introduces a cycle in the temporal serialization graph. In the affirmative case, the scheduler rejects the operation $p_i(x)$, undoes the effect of the operations of T_i, removes the inserted edge and informs the client C_i about the abort of the transaction T_i. On the other hand, $p_i(x)$ is accepted and scheduled.

Step 3. When the scheduler receives a commit request, of a transaction T_i, together with the number of operations of T_i, it verifies if all the operations of the transaction T_i have already been received and scheduled and if this still is an active transaction. In this case the commit operation will be executed and the transaction committed. If the scheduler has not received all the operations from T_i, it must delay the reply. Transaction T_i may have already been previously aborted and that the abort information has not yet arrived in the client application. In this case the abort is re-dispatched to the client. After the commit of T_i, the scheduler increments the version (timestamp) of all data items updated by T_i, as following: $\forall x$ update for T_i do $C(x) = (z, y)$ receive $(z + 1, 0)$. After this, the new values of the data items updated by T_i, together with the new timestamps, must be propagated for all the replicated servers with which the primary server has communication. If some replicated server Re_j is outside of the communication area, its identifier must be stored in log, in order for the scheduler to later try to re-dispatche this information.

Step 4. When the scheduler receives an abort request of a transaction T_i, it undoes the effect of the T_i operations, removes the edges associated with this transaction and sends the abort confirmation to client C_i.

4.4 Running Example

To illustrate the applicability and use of our proposal, we will use an electronic commerce application where several products are on sale in an electronic auction as an example. Consider that to improve data availability, get performance profits and maximize the throughput, the main database, DB_{PC}, was replicated in two others hosts (R_1 and R_2). Thus, there exists two replicated servers, each of which having a (total ou partial) database copy of DB_{PC}. This configuration is shown in figure 1.

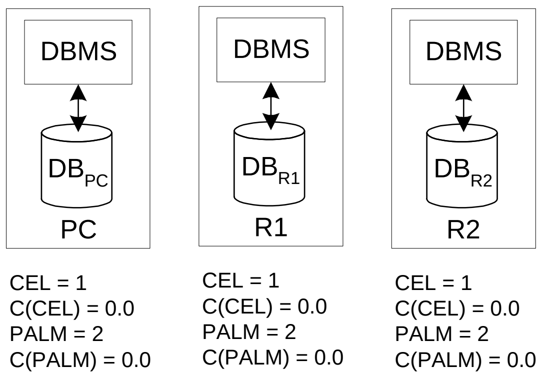

Figure 1: Replicated Environment.

Now, consider the following set of transactions, which read/write (do bidding) values of the products for sale and a schedule S_1 (figure 2), defined over the set $T = \{T_1, T_2, T_3\}$.

$T_1 : r_1(CEL)\, r_1(PALM)\, w_1(CEL, CEL + 5)\, C_1;$

$T_2 : r_2(CEL)\, w_2(CEL, CEL + 7)\, C2;$

$T_3 : r_3(PALM)\, r_3(CEL)\, w_3(CEL, CEL + 10)\, C_3;$

$$S_1 = r_1(CEL)_{R1}\ r_1(PALM)_{R1}\ w_1(CEL, CEL+5)_{R1}\ r_2(CEL)_{R1}\ w_2(CEL, CEL+7)_{R2}$$
$$r_3(PALM)_{R1}\ r_3(CEL)_{R1}\ w_3(CEL, CEL+10)_{R2}\ \ C_1\ \ C_2\ \ C_3$$

Figure 2: Schedule S_1.

We will assume that in the execution scenario presented in figure 2 the operations had been temporally executed in the presented order. The traditional serialization graph for schedule S_1 is illustrated in figure 4 (a). Observe that this graph doesn't have a cycle. Therefore, schedule S_1 could be considered conflict serializable (correct). However, if we observe the final value of the item "CEL", in copy DB_{R_1} this item has a value of 6, while in the copy DB_{R_2} and in the primary copy, this item has a value of 16. However, both values are inconsistent since this value would not be generated by any serial execution of the transactions T_1, T_2 and T_3 over DB_{PC}. Therefore it becomes necessary to revisit the replica control protocol.

Observe that this inconsistency results from the fact that the value read by the operation $r_3(CEL)_{R_1}$ does not correspond to the value written for $w_2(CEL, CEL + 7)_{R_2}$. Therefore, the value read for the operation $r_3(CEL)_{R_1}$ is previous to the value generated by the operation $w_2(CEL, CEL+7)_{R_2}$. The temporally correct schedule (S_1') is shown in figure 3, while the correct temporal serialization graph for the schedule S_1' is presented in figure 4 (b). Thus, a non conflict serializable (incorrect) schedule was considered correct. Of course, this phenomenon must be avoided because it would generate inconsistencies in the database. By the proposed protocol, the correct temporal serialization would be generated, a cycle would be identified and the transaction T_3 aborted. In this case the final value of the item "CEL" in the copy DB_{R_1} would be 6, while in the copy DB_{R_2} and in the primary copy this item would have a value of 16. The value 16 would then be propagated by the primary copy for the other copies. However, this is an consistent value. Therefore it would be generated by both the serial executions of the transactions T_1 and T_2.

$$S_1' = r_1(CEL)_{R1}\ r_1(PALM)_{R1}\ w_1(CEL, CEL+5)_{R1}\ r_2(CEL)_{R1}\ r_3(PALM)_{R1}$$
$$\boxed{r_3(CEL)_{R1}\ w_2(CEL, CEL+7)_{R2}}\ w_3(CEL, CEL+10)_{R2}\ \ C_1\ \ C_2\ \ C_3$$

Figure 3: Schedule S_1'.

Its important to note that if the packages with the information about the reads and writes operations executed for the transactions in the several servers (copies) to arrive at the primary server in a different order from the order where the operations had been executed, the proposed protocol continues generating the correct serialization graph and identifying the cycle.

Let's observe a second example. Consider the same previous configuration. Consider also the following set of transactions, which read and update the values of the products for sale:

$T_1 : r_1(CEL)\, r_1(PALM)\, w_1(CEL, CEL + 5)\, C_1;$

$T_2 : r_2(PALM)\, r_2(CEL)\, w_2(CEL, CEL + 7)\, C2;$

Now, observes schedule S_2 presented in figure 5

We will assume that in the execution scenario presented in figure 5 the operations that had been temporally executed in the presented order. The traditional serialization graph for schedule S_2 is showed in figure 7 (a). Observe that this graph doesn't present a cycle. Therefore, the schedule S_2 could be considered conflict serializable (correct). However, if we observe the final value of the item "CEL", we will see that in copy DB_{R_1} this item has a value of 6, while

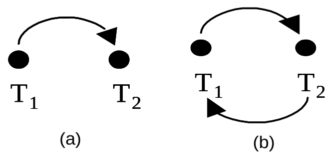

Figure 4: Serialization Graphs for the Schedules S_1 and S_1'.

$S_2 = r_1(CEL)_{R1}\ r_1(PALM)_{R1}\ w_1(CEL, CEL+5)_{R1}\ r_2(PALM)_{R1}$
$r_2(CEL)_{R2}\ w_2(CEL, CEL+7)_{R2}\ C_1\ C_2$

Figure 5: Schedule S_2.

in the copy DB_{R_2} and in the primary copy, this value is 8. However, both the values are inconsistent, whereas these values would not be generated by any serial execution of the transactions T_1 and T_2.

Observe that this inconsistency results from the fact that the value read for the operation $r_2(CEL)_{R_2}$ does not correspond to the value written for $w_1(CEL, CEL{+}5)_{R_1}$. Therefore, the value read by the operation $r_2(CEL)_{R_2}$ is prior to the value generated for the operation $w_1(CEL, CEL{+}5)_{R_1}$. The temporally correct schedule (S'_2) is shown in figure 6, while the correct temporal serialization graph for schedule S'_2 is presented in figure 7 (b). Thus, a not conflict serializable schedule (incorrect schedule) was considered correct. Of course, this phenomenon must be avoided because it would generate inconsistencies in the database. In the proposed protocol, the correct temporal serialization graph would be generated, a cycle identified and the transaction T_2 aborted. In this in case the final value of the item "CEL" in the copy DB_{R_1} and the primary copy would be 6, while in the copy DB_{R_2} this item would have a value of 1. Then, the value 6 would be propagated by the primary copy for the other copies (servers). However, this value is consistent because, it was generated by the execution of an only transaction (T_1).

$S'_2 = r_1(CEL)_{R1}\ r_1(PALM)_{R1}\ r_2(PALM)_{R1}$
$\boxed{r_2(CEL)_{R2}\ w_1(CEL, CEL+5)_{R1}}\ w_2(CEL, CEL+7)_{R2}\ C_1\ C_2$

Figure 6: Schedule S'_2.

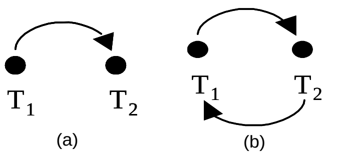

Figure 7: Serialization Graphs for the Schedules S_2 and S'_2.

4.5 Protocol Correctness

Next, we will demonstrate that the proposed algorithm ensures two basic correctness criterion: a) one-copy serializability, that is, the concurrent execution of a set of transactions on a replicated database will be equivalent to a serial execution on a same non-replicated database, and b) all the copies, eventually, will converge to one same consistent state. First, we will prove that all schedules produced by the proposed protocol are conflict serializable, that is, they ensure the data consistency in all the replicated servers.

Theorem 1. Let $TSGT_R$ be the set of all schedules on the set $T = T_1, T_2, ..., T_N$ of transactions produced by the proposed protocol and CSR the set of all the conflict serializable schedules on T. Then $TSGT_R = CSR$.

Sketch of Proof. It is easy to show that $TSGT_R \subset CSR$. We only need to observe that every schedule S produced by the proposed protocol presents an acyclic temporal serialization graph (TSG). By definition, the temporal serialization graph for S represents the conventional serialization graph, adding of temporal information to capture the correct execution order of the operations in S. In other words, if the TSG for S is acyclic, the conventional serialization graph also is. Therefore, $S \in CSR$ and consequently, $TSGT_R \subset CSR$.

To prove that $TSGT_R \supset CSR$, we need to show that every $S \in CSR$ schedule can be produced by the proposed protocol. We can show this using induction in the size of S that all operation p in S cannot generate a cycle in the TSG. For this reason operation p can be executed. As previously mentioned, the TSG represents the convention serialization graph, adding temporal information.

Now, we prove that all the copies eventually will converge to one same consistent state.

Theorem 2. If $R = Re_1, Re_2, ..., Re_N$ is the set of the replicated servers, and $DB(Re_i, t))$ the database state in the replicated server Re_i, in the time instant t, then, a position in future time t will exist where, $\forall i, j$ (with $i \neq j, 0 < i, j \leq n$) $DB(Re_i, t) = DB(Re_j, t)$ and $DB(Re_i, t)$ is consistent.

Sketch of Proof. In step 3 of the protocol, we observe that after the commit of a transaction T_i, the scheduler propagates the new value of the data items updated by T_i to all replicated servers with which the primary server has communication. If some replicated server Re_k is out of the communication area, its identification must be stored in log, since the scheduler should re-send this information. Certainly, a time instant t will exist in which there are no updates being carried out. At this instant t, $DB(Re_j, t)$, where Re_j is the primary server, is consistent and stable, and, additionally, all the replicated servers Re_k will have a connection, even if temporary, with the primary server Re_j, receiving the pending updates and updating its local copy state, making $DB(Re_k, t) = DB(Re_j, t)$. Thus, by Theorem 1, $DB(Re_k, t)$ is consistent ($\forall\ k$ com $0 < k \leq n$).

5. DISCUSSION

The mobile database replication mechanism we are proposing is optmistic (read-any/write-any model), centralized (uses a master copy to maximize the commitment process) and ensures strong consistency (by the use of serializability to guarantee data consistency). Beside using an already consolidated correctness criterion, the proposed protocol relaxes the isolation property. This property predicts that the transactions are isolated, that is, the transaction results only can be seen at the end of their execution, whether it finished successfully or not. The proposed protocol relaxes the isolation and adopts a read-any/write-any approach, allowing higher data availability. Moreover, the operations that compose the several transactions are executed only once and the servers do not need to store a log with all the executed operations. These two last properties are not guaranteed by the mechanism presented in [13]. In addition, our approach reduces the number of messages exchanged among the servers. So, our protocol looks for guarantees in savings in the communication channels use cost, in the limited power capacity of the portable computers and in its scarce memory resources.

In our approach, the server needs to keep a serialization graph (TSG) of all the active transactions, which can reduce the scalability. However, the garbage collection strategy presented in [10] can be used to reduce the graph size. Another drawback stems from the fact that the replicated servers have to send a message informing the data items which have been read and/or written to the primary server. However, such a message consists only of the item, transaction and server identifier, and one timestamp.

Similar to the approach shown in [13], a primary server is used and the replica coherence is ensured only eventually. However, in our protocol, the conflict detention is carried out through the comparison between timestamps, which consist of numerical values pairs, which provide low costs. In [13], a method is used where each write is associated with a set of validation rules and with a procedure, written in a high level interpreted language, for conflict resolution, that is, the task of detection and conflict resolution is transferred to the programmers.

Finally, the assumption in Theorem 2 that there will be a time instant in which the system will be in the absence of updates, despite seeming strong, is common in other replication mechanisms, as for example, in the approach presented in [13, 12].

6. CONCLUSIONS

In this paper we present a new mechanism that guarantees the replicated data consistency (one-copy serializability) and the replica convergence to the same consistent final state (eventual consistency), in mobile computing environments. This protocol explores the temporal information about the moment where a replicated database object was read or updated and uses the serializability as correctness criterion. It's important to note that serializability already became a standard, and, therefore, practically all existing DBMS in the market implement this model. Besides using a consolidated correctness criterion, the proposed protocol relaxes the isolation property and adopts the read-any/write-any approach, allowing higher data availability. In this mechanism, the operations that compose the several transactions are executed only one time and the servers do not need to store a log with all executed operations. Moreover, our approach reduces the number of messages exchanged between the servers. All this will ensure a savings in the communication channels cost, in the limited portable computers energy capacity and in its scarce memory resources. We intend to extend the proposed protocol in order to allow that the isolation level be adapted to the conditions of the mobile environment in future studies through the choice of one amongst the four standardized ANSI/ISO SQL 92 [1] isolation levels, and make the item updates propagation in a peer-to-peer (P2P) manner possible.

7. REFERENCES

[1] A. Adya, B. Liskov, B. O'Neil, and P. O'Neil. *Generalized Isolation Level Definitions*. In proceedings of the IEEE International Conference on Data Engineering, San Diego, CA, March, 2000.

[2] J. Barreto and P. Ferreira. *An Efficient and Fault-Tolerant Update Commitment Protocol for Weakly Connected Replicas*. Proceedings of the EUROPAR, Lisboa, Portugal, 2005.

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

[4] M. A. Casanova. *The Concurrency Problem of Database Systems*. In Lectures Notes in Computer Science, 116, 1981.

[5] U. Cetintemel, P. Keleher, and M. Franklin. *Support for Speculative Update Propagation and Mobility in Deno*. Proceedings of the The 21st IEEE International Conference on Distributed Computing Systems (ICDCS'01), 2001.

[6] D. K. Gifford. *Weighted Voting for Replicated Data*. Proceedings of the Seventh Sysmposium on Operating Systems Principles, 1979.

[7] R. Golding and D. Long. *Modeling replica divergence in a weak-consistency protocol for global scale dirstibuted data bases*. Technical Report UCSC-CRL-93-09, 1993.

[8] P. Keleher. *Decentralized replicated-object protocols*. In: Proc. of the 18th Annual ACM Symp. on Principles of Distributed Computing (PODC'99), 1999.

[9] R. Maya and L.Anthony. *Decentralized Weighted Voting for P2P Data Managemento*. Proceedings of the 3rd ACM International Workshop on Data Engineering for Wireless and Mobile Access, San Diego, CA, USA, (MobiDe '03), 2003.

[10] J. M. Monteiro and A. Brayner. *Controlling Concurrency in Mobile Computing Environments with Broadcast-based Data Dissemination*. Proceedings of the EUROPAR, Lisboa, Portugal, 2005.

[11] Y. Saito and M. Shapiro. *Optimistic Replication*. ACM Computing Surveys, Vol. V, No. 3, 2005.

[12] D. Terry, A. Demers, K. Petersen, M. Spreitzer, and M. Theimer. *Flexible Update Propagation for Weakly Consistent Replication*. 1997.

[13] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, B. Welch, and C. Hauser. *Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System*. 1995.