Pre-serialization of long running transactions to improve concurrency in mobile environments

Angelo Chianese #, A. d'Acierno *, V. Moscato #, A. Picariello #

#*Dipartimento di Informatica e Sistemistica, University of Naples*
via Claudio 21, 80125, Naples, Italy
{angchian,vmoscato,picus}@unina.it

**ISA-CNR*
via Roma 52, 83100,Avellino, Italy
dacierno.a@isa.cnr.it

***Abstract*— Transaction management in different application contexts is still a challenging task. In this paper we propose a novel method in order to improve concurrency of particular kind of transaction, known as *long running* transactions. Differently from other techniques presented in the literature, we design a sort of hybrid approach between optimistic and pessimistic concurrency models. From one hand, our basic idea consists of handling frequent disconnections or inactivity periods of a generic transaction during its life-cycle and, from the other one, we consider the semantics related to operations produced by transactions. First, our solution avoids an indefinite or long resource locking due to disconnecting (or idle) transactions or a high rate of preventive aborts; eventually, a transaction "semantic compatibility" is exploited in order to increase the concurrency reconcilable operations on the same resources. To these purposes, we have implemented a middleware with the aims of emulating a transactional scheduling, and several experiments have been carried out.**

I. Introduction

Traditional transaction management techniques are no longer appropriate for a variety of application contexts [2], [1], [16]. In particular, the lengthy transmission delay of some networks, frequent and unpredictable disconnections, long inactivity periods of users could affect transaction duration, generating *long running transactions*. In this framework several problems, such as low concurrency rate, deadlocks and starvation, handling of disconnections, and so on, have to be solved [13].

Classical *pessimistic* solutions for concurrency control based on *Two Phase Locking* (2PL) are not suitable: in fact, the long duration of such a kind of transactions forces a long time resource locking due to disconnecting or idle transactions, or, in the opposite, a high rate of preventive aborts.

To limit such problems, *optimistic* approaches allow different transactions to immediately and concurrently operate on the various resources or by relaxing DBMS locking policies or by replicating the shared data on user devices [11]. Anyway such approaches could cause the management of a high number of rollback operations on updated data when a high rate of transaction conflicts occurs.

One of the first paper concerning this kind of approach was proposed by [6], where an "optimistic scheme" is introduced, resolving the synchronization of read-write conflicts using "dummy" locks. Even if the dummy locks are effectively long-term locks, they do not block the execution of transactions, providing a suitable deadlock-free locking technique, obtained pre-ordering the data items. A similar interesting approach is the *O2PL-MT* (Optimistic 2PL for Mobile Transactions) [7] that extends the optimistic version of 2PL (O2PL) to mobile environments. In such a solution "read-blocks" are allowed under request to transactions on distributed resources, while "write-blocks" are differed until commit time. In [3] a concept of "compensation" that tolerates *weak or semantic atomicity* is proposed. Another more recent technique is the *ASGT* (Active Serialization Graph Technique) [19], in which read-operations on a given resource do not obstruct write-operations of a generic transaction. By means of an apposite serialization graph the transaction involved in not-serializable schedules are aborted to reduce conflicts-queue and rollback costs.

Alternative techniques suggest a relaxation of consistency property. To improve concurrency rate, transactions are allowed to operate on inconsistent copies of data items. Prabhu and Kumar [13] proposed an approach in which transactions first perform their operations (a maximum number of changes in given timeout is permitted) on local data replica successively reconciliated with the data on the DBMS. Pitoura and Bhargava [12] introduce a *Clustering* strategy, in which distributed database data are subdivided in clusters on the base of their semantic or geographical position and two kinds of data replica from consistency point of view, "strict" and "weak", are used. In a first phase disconnected transactions execute the commit on local weak cluster-data, while in the second phase the local data are reconciliated with strict data, if the case, generating the abort or rollback. Eventually, in [17] the local data, on which disconnected transactions will operate, are subdivided on the base of semantic criteria and reconciliating operation are if necessary exploited to guarantee consistency.

Other techniques focus their attention on isolation and atomicity [14]. A first example is the *Pre-Serialization* proposal [5] that preserves isolation for mobile and disconnected long running transactions by associating themselves to "site transactions" and by using serialization graphs to ensure serializability. A second example is the *IOT* (Isolation Only

Transactions) [10], in which transactions are subdivided in "not-disconnected" and "disconnected" transactions: for the first ones commit is immediately executed, for the second ones the migration to a "pending" state is necessary and the commit is executed only after a validation phase.

Most of the progress on the concurrency control theory and 2PL protocols, have been also obtained in the *Real Time Data Base* (RTDB) realm, characterized by having long and complex transaction. A number of studies have been done on this subject. In [4], for example, different algorithms and protocols are suggested to schedule transactions according to their priorities. In [9], [8], a nested transaction model for mobile real-time transactions is proposed, together with a divergence control lock model based on prudent ordered sharing and a *Check-Out/Check-In* protocol supporting disconnections: as a result, *DC/POS-PAI-2PL* two-phase lock strategy integrating these methods for mobile real-time transactions is eventually proposed.

An other important area of related work is on nested transactions. Among the more recent proposals, Vingralek et al. [15] provide a flexible concurrency control mechanisms for nested transactions that exploits semantics of transaction operations.

In this paper we propose a novel method in order to improve concurrency of long running transactions. Differently from other techniques presented in the literature, we design a hybrid approach between optimistic and pessimistic models. Our basic idea consists of a "re-visitation" of the 2PL protocol, in order to handle frequent disconnections or inactivity periods of transactions, considering the semantics associated to the related operations: in this way, we try to overcome both long time resource locking and a high rate of preventive aborts; moreover, exploiting a transaction "semantic compatibility" based on the Weihl's theory [18], we increase the overall concurrency rate. In our approach we take into account that in a lot of "real world" applications is not important to exactly track the dynamic behavior of each "real entity" to allow the correct execution of a transaction. This observation is surely valid only for certain kinds of transaction operations.

The paper is organized as follows. Section II describes a motivating example for our paper. Section III outlines the architecture of the proposed system. Sections IV and V describe the model and the algorithms for transaction management. The aim of section VI is to demonstrate, by a model evaluation, the possible advantages of our approach and a comparison with 2PL protocol. Open problems, on going work and concluding remarks are discussed in section VII.

II. Motivating Scenario

In order to understand the main aspects that our work is addressing, we will discuss a motivating example, showing the problems and discussing the weakness of traditional solutions.

Let us consider an hypothetical agency which sells, via web, *personalized* package tours for visiting museums: a user buys flight tickets, makes hotel reservation, rents a car and reserves tickets for museums. The different information (flight tickets, hotel info, etc...) are contained in a unique database and mobile clients may be used in a network with frequent disconnections (e.g. wireless network). A user u_i, in general, may perform the following actions: (i) select a flight by setting a departure date, company, and so on; (ii) book the chosen flight; (iii) check for hotel room availability; (iv) make a hotel reservation; (v) make a ticket reservation for a museum; (vi) rent a car;...and so on.

Clearly, these actions are iteratively performed: when users do not find an available hotel in a certain town, they can decide to change *return* date or the scheduling of the tour. When they are happy with all the selected options, they *commit* the whole package tour. From the DBMS point of view, this process can be simply schematized as follows:

```
begin transaction T_i
while UserNotHappy do
   select FreeTickets from Flight where some_conditions
   if the case... then
      update Flight set FreeTickets = FreeTickets -1
      where some_conditions
   end if
   select FreeTickets from Museum where some_conditions
   if the case... then
      update Museum set FreeTickets = FreeTickets -1
      where some_conditions
   end if
   select FreeCars from Car where some_conditions
   if the case... then
      update Car set FreeCars = FreeCars -1
      where some_conditions
   end if
end while
commit
end transaction T_i
```

Using the traditional 2PL strategy, we can assume that $\mathcal{T}_i$ requests a *read-lock* on Flight.FreeTickets and then moves to a write-lock, when the user reserves the ticket. In this case, if another user starts a transaction $\mathcal{T}_j$ thus acquiring the read-lock on the same Flight.FreeTickets field, a deadlock can occur and it can be solved aborting $\mathcal{T}_i$ and/or $\mathcal{T}_j$. When the number of requests increases, the number of aborted transactions could become unacceptable. Alternatively, but still using a 2PL strategy, we can assume that we know *the semantic of the transactions*: in this case we can grant the write-lock to $\mathcal{T}_i$ on Flight.FreeTickets. If the user does not quickly decide to commit or abort the whole operation (e.g., due to network disconnection, inactivity periods, etc...), a long time write-lock occurs, and another user accessing to the same resource has to wait.

Another widely used strategy consists of: (i) imposing precise constraints on important resources (for example, Flight.FreeTickets $\geq$ 0) and (ii) assuming that each user operation is temporarily *freezed* and the whole transaction will be executed when the user commits. Using such a strategy, deadlocks become highly unfrequent, because both read and write locks are released as quick as possible and the system exploits the highest possible concurrency level among the users. The drawback of this method is that no locks are held

while the users are performing their actions, thus resulting in boring, and sometimes dramatic, changes to several attributes (e.g., no more flight tickets available and the whole journey has to be replanned!).

These kinds of problems are avoided in the proposed strategy. In particular, the main features of our approach are: (i) to consider the *semantic* of operations generated by transactions (ensuring a high degree of concurrency among users for a certain kind of operations); (ii) to introduce the concept of *sleeping* transaction, where a sleeping transaction is a disconnected or idle transaction that does not have to be aborted, but that could reconnect and try to end its work.

III. System Architecture

The proposed system offers a set of services to manage transactions and is based on the following layers:

- *Data Layer*: this layer offers the functionalities to manage and store application data that are contained in a Local DataBase System (LDBS), a traditional relational DBMS.
- *Middleware Layer*: it is the core part of the entire architecture. It represents our transactional manager middleware. It accepts user requests in terms of transactions, generated by mobile or wired devices and then processes them in according to the transaction concurrency management model, as will be described in the next section. In particular, it manages: (i) transaction concurrency on the base of their *semantic compatibility*, (section IV); (ii) possible transaction deadlocks or starvation; (iii) disconnections or inactivities of user clients. Moreover, this module is wired connected to the LDBS server in order to manage data on the local database.
- *Application Layer*: at this layer the user applications coming from mobile or wired devices generate the transactions that have to be scheduled.

In our approach the transactions are first managed by a *Global Transaction Manger* (*GTM*-middleware), operating in a *virtual context* on a set of virtual data – that are special images of database data, "local" to each transaction. At commit time, a special transaction, called *Secure System Transaction* (*SST*), is then generated by the GTM to report the effective changes from the virtual copies to the LDBS. At the middleware layer only isolation and atomicity properties have to be ensured, in the opposite, consistency and duration are delegated to data layer DBMS. This approach improves the concurrency, allowing, under some special conditions, a shared lock on virtual data; moreover, it is possible to manage disconnections (due to network problems) and user-inactivity, trying to resume the previous work performed by a user transaction.

IV. The Proposed Model

Our model of computation can be seen as an extension of Weihl's [18] approach.

Let us consider two kinds of entities, *transactions* and *objects*. A transaction (that we denote with the symbols A, B, C, etc...) accesses (reads and modifies) the object state, by means of a set of operations that each object (denoted with the symbols X, Y, Z, etc...) provides, and vice-versa, each object can modify the state of a given transaction.

Generally speaking, a transaction can assume different *operating states* during its life cycle. In particular, the set of possible states that a transaction can assume is: *Active* – a transaction is in such a state when it is normally running –, *Waiting* – a transaction is in such a state when it is waiting for a lock –, *Sleeping* – a transaction is in such a a state when it has been inactive for a a given period of time –, *Committing* – a transaction is in such a state if the user has requested the commit and the related *SST* has not yet been terminated –, *Aborting* – a transaction is in such a state when its job has to be aborted, *Committed* – a transaction is in such a state when the related *SST* has been performed –, *Aborted* – a transaction is in such a state if its job has been aborted.

The global state of a given transaction A is defined by the following information: $\mathtt{A}_{state}$ contains the operating state of the transaction; $\mathtt{A}_{temp}$ contains, for each object X accessed by the transaction, a *replica* of the object values on which all the transaction operations will be operating; $\mathtt{A}_{t_{sleep}}$ contains the *time* in which the transaction has become sleeping; $\mathtt{A}_{t_{wait}}$ contains, for each object X, the *arrival time* of the transaction in the related object *wait-queue*.

We also assume that the operation semantics in a transaction is *a-priori* known, so that we can associate to the transactions a set of *classes of operation*, i.e. read, insert, delete and update.

Each object can be seen as an abstract data type, characterized by a *state* and a *set of methods* that can modify such a state. In particular, its state is defined by means of the following information: $\mathtt{X}_{permanent}$ contains the committed value (the set of committed values for each data member, if X is a structured type) of the object; $\mathtt{X}_{pending}$ is the set of transactions – associated to their class of operations (operation type and object data member) – that have obtained the grant to modify the values of the object X; $\mathtt{X}_{waiting}$ is the set of transactions – associated to their class of operations – that are waiting for accessing the object X; $\mathtt{X}_{committing}/\mathtt{X}_{committed}$ is the set of transactions that are going to perform/have performed the commit on the DBMS; $\mathtt{X}_{aborting}$ is the set of transactions that are trying to perform an abort of their operations; $\mathtt{X}_{sleeping}$ contains the set of transactions, operating on X, that are sleeping; $\mathtt{X}_{read}$ contains, for each transaction operating on X, the value of X – as retrieved from the database – before any kind of modifications; $\mathtt{X}_{new}$ contains, for each transaction operating on X, the value of X that will be stored in the database when the commit will be accepted; $\mathtt{X}_{tc}$ contains, for each transaction that has performed the commit on X, the commit time.

In a given time a transaction can access to only a component of a given object. In particular, each transaction will operate – for each object involved in its execution – on a set of virtual data $\mathtt{A}_{temp}$, and only at commit time the results of execution will be stored in the DBMS. The computation is assumed to be *event-based*, and we will be focusing on those events regarding the interaction between transactions and objects. In particular, we consider the following kinds of events of interest:

- **Start**, $\langle begin, \mathtt{A}\rangle$, indicates that the transaction A is started.
- **Invocation**, $\langle op, \mathtt{X}, \mathtt{A}\rangle$, indicates that A requests the grant for a set of operations *op* (of the same class) on X (or on its component, if X is a structured type).
- **Local Commit/Global Commit**, $\langle commit, \mathtt{X}, \mathtt{A}\rangle$ and $\langle commit, \mathtt{A}\rangle$: the first one is the event informing the object X (modified by A) that the transaction is requesting a commit; the second one is the global commit of A.
- **Local Abort/Global Abort**, $\langle abort, \mathtt{X}, \mathtt{A}\rangle$ and $\langle abort, \mathtt{A}\rangle$: the firs one is the event informing the object X (modified by A), that the transaction is requesting an abort, the second one is the global abort of A.
- **Local Sleep/Global Sleep**, $\langle sleep, \mathtt{X}, \mathtt{A}\rangle$ and $\langle sleep, \mathtt{A}\rangle$: the first one is the event informing the object X that the transaction A is sleeping; the second one is the state-transition of A to the *Sleeping* state.
- **Local Awakening/Global Awakening**, $\langle awake, \mathtt{X}, \mathtt{A}\rangle$ and $\langle awake, \mathtt{A}\rangle$: the first one is the event informing the object X that A is awaking; the second one is the state-transition of A from the *Sleeping* state.
- **Unlock**, $\langle unlock, \mathtt{X}\rangle$ is the event informing the transactions that the object X does'nt have more pending operations.

We assume that: for a given transaction at most one pending invocation of a single object data member at any time is permitted; the invocation and commit events are allowed if and only if a transaction is in the *Active* state; after performing its operations, a transaction can commit or abort at one or more objects, but not both for a given object; after a commit/abort on a given object no operation is allowed for the current transaction.

This allows to guarantee that the events sequence generated by transactions on a given object X, is *well-formed* and all atomicity properties can be ensured, as requested by [18]. We thus model the serial specification of an object X or a transaction A (acceptable behavior of X and A in a sequential environment), using a language defined on the state machines $\mathcal{S}(\mathtt{X})$ and $\mathcal{S}(\mathtt{A})$.

On these machines, a transition function able to determine the new state s depending on the previous state s' is defined, together with the event generated by a transaction or object, $\mathcal{T} : (s', event) \rightarrow s$.

Now we are in the position of introducing the concept of compatibility among transaction operations (i.e. invocation events), that is a specification of *Weihl's forward commutativity* [18].

Definition 1 (Transaction Operations Compatibility): Two invocation events ω_1 and ω_2 generated by two generic transactions, A and B, are compatible iff $\forall$ state s of the object X:

1) ω_1 and ω_2 are reffered to the same object data member,
2) $\mathcal{T}(\mathcal{T}(s, \omega_1), \omega_2) = \mathcal{T}(\mathcal{T}(s, \omega_2), \omega_1)$ $\mathcal{T}(\mathcal{T}(s, \omega_1), \omega_2) \neq \perp$,
3) there exists a *reconciliating algorithm* that can determine, knowing the object and transaction states, the final correct value to be stored in the database at the transaction's commit event.

Class of operations	Compatibilities
Read	All classes
Insert/Delete	No classes
update with assignment	Read
update with add/sub operations	Addition/Substratcion, Read
update with mult/div operations	Multiplication/Division, Read

TABLE I
CLASS OF COMPATIBILITIES

The discussed concept of compatibility can be relaxed considering the "logical dependence" among object data members. In other terms only transaction operations on logically dependent items (e.g. quantity and price of a given product) can generate a conflict, while operations on not-logical dependent data members are compatible.

Addition/subtraction or multiplication/division of X (where the object is supposed to be of atomic type) by a constant value (i.e., $\mathtt{X} = \mathtt{X} \pm \mathtt{c}$, $\mathtt{X} = \mathtt{X} \cdot c$, or $\mathtt{X} = \frac{\mathtt{X}}{\mathtt{c}}$ with $\mathtt{c} \neq 0$) are examples of compatible operations.
A reconciliation algorithm for addition/subtraction is the following:

$$\mathtt{X}^{\mathtt{A}}_{new} = \mathtt{A}^{\mathtt{X}}_{temp} + \mathtt{X}_{permanent} - \mathtt{X}^{\mathtt{A}}_{read} \tag{1}$$

A reconciliation algorithm for multiplication/subtraction is:

$$\mathtt{X}^{\mathtt{A}}_{new} = \frac{\mathtt{A}^{\mathtt{X}}_{temp}}{\mathtt{X}^{\mathtt{A}}_{read}} \cdot \mathtt{X}_{permanent} \tag{2}$$

We observe that assignment operation (i.e., $\mathtt{X} = c$) is not compatible with any kind of operations. We also note that the data locked by a certain transaction can be modified by other compatible operations pertaining to a different transaction. In the following, for the sake of simplicity, we will assume no difference between read operations finalized to update, and write operations. Table 1 schematizes the class of operations considered in our model and their related compatibilities. Table 2 shows an example of reconciliation between two transactions performing simple addition operation on the same object.

We explicitly note that not all the possible sequences of events are allowed. In fact, we assume that the sequence of events $\mathcal{E}$ are submitted to the following constraints: (i) a transaction A can execute on a given object component only compatible operations; (ii) a transaction A can commit or abort, in $\mathcal{E}$ but not both; (iii) a transaction A cannot commit if it is waiting for an invocation and and cannot invoke any operations after it commits; (iv) a sleeping transaction A has to be awaken before to continue its execution. These restrictions on transactions are intended to model some typical uses of transactions: note that, constraints (ii) and (iii) are traditional constraints of a transactional environment; (i) and (iv) are specific of our model. In particular, (i) is justified by the considered scenarios in which transaction during their

A code	**B code**	$X_{permanent}$	X^{A}_{read}	A^{X}_{temp}	X^{A}_{new}	X^{B}_{read}	B^{X}_{temp}	X^{B}_{new}
begin	-	100	-	-	-	-	-	-
read X	begin	100	100	100	-	-	-	-
X = X+1	read X	100	100	100	-	100	100	-
write X	X=X+2	100	100	101	-	100	100	-
X = X+3	write X	100	100	101	-	100	102	-
write X	-	100	100	104	-	100	102	-
req commit	-	100	100	104	104	100	102	-
commit	req commit	104	-	-	-	100	102	106
-	commit	106	-	-	-	-	-	-

TABLE II
EXAMPLE OF RECONCILIATIONS

execution usually perform just one kind of operations on the data.

Eventually, before discussing the proposed transaction management algorithms, we give the following definition.

Definition 2 (Transaction Conflicts): Transactions A and B are *in conflict* on X, $(\mathtt{A}, \mathtt{B}) \in CONFLICT_{\mathtt{X}}$, if A is operating on X and B requests to perform an operation that is not compatible with the set of current operations of A, or vice-versa.

V. CONFLICT-BASED TRANSACTION MANAGEMENT

The proposed GTM implements a particular *locking-based* protocol and its focus is the management of transactions conflicts over an object X providing a sort of pre-serialization of transactions to improve concurrency rate. The main purpose of GTM is to handle both *synchronous* and *asynchronous* events – such as invocations, commit or abort, transaction sleep and awake, object unlocks – thus exploiting semantic and compatibility between transaction operations.

In the following we show the transactions and the objects' behaviors for each considered event. The global transaction manager is, in a certain way, a sort of controller for the state machines that manages the transaction conflicts on the various database objects, thus allowing a pre-schedule of transactions. Summing up, when a transaction is instantiated, its state is *Active*. After a lock request on a given object, the transaction can remain in the same state or migrate in another state, depending on the following factors: current state of the object; transaction operations to be executed; operations of transactions that have locked the object. Once obtained a lock on a given resource, a generic transaction will use the replicated data $\mathtt{A}^{\mathtt{X}}_{temp}$ to perform its operations. Only at commit event, the reconciliated data will be updated on the database.

It is clear how in this model a *Sleeping* transaction can awake (we suppose that the sleep was due to user inactivity or to disconnection for a temporary network fault), and tries to finish its work if there were not incompatible operations that have requested the lock on the same resources and operated on the same data.

As discussed, from the GTM point of view only atomicity and isolation properties have to be ensured, while consistency and durability are guaranteed by generating secure system transactions that will be managed by a classical DBMS. So at the GTM level, classical *lost update*, *uncommitted dependency* and *inconsistent analysis* are not solved but admissible.

It easy to demonstrate that, assuming no constraint on the database objects, the schedule generated by GTM is *serializable*. The presence of reconciliation algorithms assures that the obtained schedule is the same of a typical locking-based concurrency control method as 2PL. In particular the conditions that ensure serialiazability are: the compatible operations operate on virtual data and during transaction execution are

Algorithm 1 GTM behavior for $\langle begin, \mathtt{A}\rangle$

Postcondition:
$\mathtt{A}_{state} = Active$

Algorithm 2 GTM behavior for $\langle op, \mathtt{X}, \mathtt{A}\rangle$

Precondition (compatible operations):
$\mathtt{A}_{state} = Active$
$(\mathtt{A}, \mathtt{B}) \notin CONFLICT_X \ \forall \mathtt{B} \in (\mathtt{X}_{pending} - \mathtt{X}_{sleeping}) \cup \mathtt{X}_{committing}$
Postcondition:
$\mathtt{X}_{pending} = \mathtt{X}_{pending} \cup (\mathtt{A}, op)$
$\mathtt{X}^{\mathtt{A}}_{read} = \mathtt{X}_{permanent}$
$\mathtt{A}^{\mathtt{X}}_{temp} = \mathtt{X}_{permanent}$
Precondition (some not-compatible operations):
$\mathtt{A}_{state} = Active$
$\exists \mathtt{B} \in (\mathtt{X}_{pending} - \mathtt{X}_{sleeping}) \cup \mathtt{X}_{committing}$ s.t. $(\mathtt{A}, \mathtt{B}) \in CONFLICT_X$
Postcondition:
$\mathtt{A}_{state} = Waiting$
$\mathtt{A}_{twait} = Current\_Time$
$\mathtt{X}_{waiting} = \mathtt{X}_{waiting} \cup (\mathtt{A}, op)$
$\mathtt{A}^{\mathtt{X}}_{temp} = \perp$

Algorithm 3 GTM behavior for $\langle commit, \mathtt{X}, \mathtt{A}\rangle$

Precondition:
$\mathtt{A}_{state} = Active \vee \mathtt{A}_{state} = Committing$
$\mathtt{A} \in \mathtt{X}_{pending}$
$\nexists \mathtt{B} \in \mathtt{X}_{committing}$ s.t. $\mathtt{B} \neq \mathtt{A}$
Postcondition(ϱ is the procedure implementing the reconc. alg.):
$\mathtt{A}_{state} = Committing$
$\mathtt{X}_{committing} = \mathtt{X}_{committing} \cup (\mathtt{A}, op)$
$\mathtt{X}^{A}_{new} = \varrho(\mathtt{X}^{A}_{read}, \mathtt{A}^{X}_{temp}, \mathtt{X}_{permanent})$
$\mathtt{A}^{\mathtt{X}}_{temp} = \perp$
$\mathtt{X}^{A}_{read} = \perp$
$\mathtt{X}_{pending} = \mathtt{X}_{pending} - (\mathtt{A}, op)$

Algorithm 4 GTM behavior for $\langle commit, \mathtt{A} \rangle$

Precondition:
$\mathtt{A}_{state} = Committing$
$(\mathtt{A} \in \mathtt{X}_{committing})$ ($\forall \mathtt{X}$ involved in $\mathtt{A}$ execution)
$\mathtt{X}^{A}_{new} \neq \perp$ $\forall \mathtt{X}$ involved in $\mathtt{A}$ execution
Postcondition:
$\mathtt{X}_{permanent} = \mathtt{X}^{A}_{new}$ $\forall \mathtt{X}$ involved in $\mathtt{A}$ execution
$\mathtt{X}^{A}_{new} = \perp$ $\forall \mathtt{X}$ involved in $\mathtt{A}$ execution
$\mathtt{A}_{state} = Committed$
$\mathtt{A}_{twait} = \perp$
$\mathtt{A}_{tsleep} = \perp$
($\forall \mathtt{X}$ involved in $\mathtt{A}$ execution)$(\mathtt{X}_{committing} = \mathtt{X}_{committing} - (\mathtt{A}, op))$
($\forall \mathtt{X}$ involved in $\mathtt{A}$ execution)$(\mathtt{X}_{committed} = \mathtt{X}_{committed} \cup (\mathtt{A}, op))$
($\forall \mathtt{X}$ involved in $\mathtt{A}$ execution)$\mathtt{X}_{tc} = Current\_Time$

Algorithm 5 GTM behavior for $\langle abort, \mathtt{X}, \mathtt{A} \rangle$

Precondition:
$\mathtt{A}_{state} = Active \vee \mathtt{A}_{state} = Aborting \vee \mathtt{A}_{state} = Waiting$
$\mathtt{A} \in \mathtt{X}_{pending} \vee \mathtt{A} \in \mathtt{X}_{waiting}$
Postcondition:
$\mathtt{A}_{state} = Aborting$
$\mathtt{X}_{aborting} = \mathtt{X}_{aborting} \cup \mathtt{A}$
$\mathtt{A}^{X}_{temp} = \perp$
$\mathtt{X}^{A}_{read} = \perp$
$\mathtt{X}^{A}_{new} = \perp$
$\mathtt{X}_{pending} = \mathtt{X}_{pending} - (\mathtt{A}, op)$

Algorithm 6 GTM behavior for $\langle abort, \mathtt{A} \rangle$

Precondition:
$\mathtt{A}_{state} = Aborting$
$\mathtt{X}^{A}_{new} = \perp$ ($\forall \mathtt{X}$ involved in $\mathtt{A}$ execution)
$\mathtt{X}^{A}_{read} = \perp$ ($\forall \mathtt{X}$ involved in $\mathtt{A}$ execution)
$\mathtt{A}^{X}_{temp} = \perp$ ($\forall \mathtt{X}$ involved in $\mathtt{A}$ execution)
Postcondition:
$\mathtt{A}_{state} = Aborted$
$\mathtt{A}_{twait} = \perp$
$\mathtt{A}_{tsleep} = \perp$
($\forall \mathtt{X}$ involved in $\mathtt{A}$ execution) $(\mathtt{X}_{aborting} = \mathtt{X}_{aborting} - \mathtt{A})$

Algorithm 7 GTM behavior for $\langle sleep, \mathtt{X}, \mathtt{A} \rangle$

Precondition:
$\mathtt{A}_{state} = Sleeping$
$\mathtt{A} \notin \mathtt{X}_{sleeping}$
Postcondition:
$\mathtt{X}_{sleeping} = \mathtt{X}_{sleeping} \cup \mathtt{A}$

Algorithm 8 GTM behavior for $\langle sleep, \mathtt{A} \rangle$

Precondition (Ξ is a an oracle that returns TRUE if $\mathtt{A}$ is sleeping):
$\mathtt{A}_{state} = Active \vee \mathtt{A}_{state} = Waiting$
$\Xi(\mathtt{A})$
Postcondition:
$\mathtt{A}_{state} = Sleeping$
$\mathtt{A}_{tsleep} = Current\_Time$

Algorithm 9 GTM behavior 1 for $\langle awake, \mathtt{X}, \mathtt{A} \rangle$

Precondition (no conflicts during sleeping-time):
$\mathtt{A}_{state} = Sleeping$
$\mathtt{A} \in \mathtt{X}_{waiting}$
$(\mathtt{A}, \mathtt{B}) \notin CONFLICT_X$ ($\forall \mathtt{B} \in \mathtt{X}_{pending} \cup \mathtt{X}_{committing}$)
$(\mathtt{A}, \mathtt{B}) \notin CONFLICT_X$ ($\forall \mathtt{B} \in \mathtt{X}_{committed}$ s. t. $\mathtt{X}^{\mathtt{B}}_{tc} > \mathtt{A}_{tsleep}$)
Postcondition:
$\mathtt{X}_{sleeping} = \mathtt{X}_{sleeping} - \mathtt{A}$
$\mathtt{X}_{waiting} = \mathtt{X}_{waiting} - (\mathtt{A}, op)$
$\mathtt{X}_{pending} = \mathtt{X}_{pending} \cup (\mathtt{A}, op)$
$\mathtt{X}^{\mathtt{A}}_{read} = \mathtt{X}_{permanent}$
$\mathtt{A}^{\mathtt{X}}_{temp} = \mathtt{X}_{permanent}$
Precondition (no conflicts during sleeping-time):
$\mathtt{A}_{state} = Sleeping$
$\mathtt{A} \notin \mathtt{X}_{waiting}$
$(\mathtt{A}, \mathtt{B}) \notin CONFLICT_X$ ($\forall \mathtt{B} \in \mathtt{X}_{pending} \cup \mathtt{X}_{committing}$)
$(\mathtt{A}, \mathtt{B}) \notin CONFLICT_X$ ($\forall \mathtt{B} \in \mathtt{X}_{committed}$ s. t. $\mathtt{X}^{\mathtt{B}}_{tc} > \mathtt{A}_{tsleep}$)
Postcondition:
$\mathtt{X}_{sleeping} = \mathtt{X}_{sleeping} - \mathtt{A}$
Precondition (conflicts during sleeping-time):
$\mathtt{A}_{state} = Sleeping$
$(\exists \mathtt{B} \in \mathtt{X}_{pending} \cup \mathtt{X}_{committed} \cup \mathtt{X}_{committing})$ s.t. $(\mathtt{A}, \mathtt{B}) \in CONFLICT_X$
$(\exists \mathtt{B} \mathtt{X}_{committed})$ s. t. $\mathtt{X}^{\mathtt{B}}_{tc} > \mathtt{A}_{tsleep} \wedge (\mathtt{A}, \mathtt{B}) \in CONFLICT_X$
Postcondition:
$\mathtt{A}_{state} = Aborted$
$\mathtt{A}_{tsleep} = \perp$
$\mathtt{A}_{twait} = \perp$
($\forall \mathtt{X}$ involved in $\mathtt{A}$ execution) $\mathtt{A}^{X}_{temp} = \perp$
($\forall \mathtt{X}$ involved in $\mathtt{A}$ execution) $\mathtt{X}_{sleeping} = \mathtt{X}_{sleeping} - \mathtt{A}$
($\forall \mathtt{X}$ involved in $\mathtt{A}$ execution) $\mathtt{X}_{waiting} = \mathtt{X}_{waiting} - \mathtt{A}$
($\forall \mathtt{X}$ involved in $\mathtt{A}$ execution) $\mathtt{X}_{pending} = \mathtt{X}_{pending} - \mathtt{A}$
($\forall \mathtt{X}$ involved in $\mathtt{A}$ execution) $\mathtt{X}^{A}_{read} = \perp$
$\forall \mathtt{X}$ involved in $\mathtt{A}$ execution) $\mathtt{X}^{A}_{new} = \perp$

Algorithm 10 GTM behavior for $\langle awake, \mathtt{A} \rangle$

Precondition:
$\mathtt{A}_{state} = Sleeping$
($\forall \mathtt{X}$ involved in $\mathtt{A}$ execution) $\mathtt{A} \notin \mathtt{X}_{sleeping}$
Postcondition:
$\mathtt{A}_{state} = Active$
$\mathtt{A}_{twait} = \perp$

not able to modify the database data; the effective commit is managed by LDBS by a classical 2PL protocol; the execution order of compatible operations does not the affect the final values of the database data.

VI. Experimental Results

A. Model Evaluation

In this subsection we report several experiments finalized to theoretically evaluate the transaction execution time and the abort percentage of sleeping transactions of our model respect to the classical 2PL. In particular, the first parameter has been evaluated at the variation of: (i) number of transaction conflicts, (ii) number of not compatible transaction operations. Let c and τ_e be respectively the number of transaction conflicts and the execution time in the ideal condition (no conflicts are verified) of a single transaction, we assume that the 2PL average execution time is given by the following equation:

Algorithm 11 GTM behavior for unlock event $\langle unlock, \mathtt{X} \rangle$

Precondition:
$\mathtt{X}_{pending} = \perp$
Postcondition:
$(\forall\ \mathtt{A} \in \vartheta(\mathtt{X}_{waiting} - \mathtt{X}_{sleeping}))(\mathtt{A}_{state} = Active)$
$(\forall\ \mathtt{A} \in \vartheta(\mathtt{X}_{waiting} - \mathtt{X}_{sleeping}))(\mathtt{X}_{pending} = \mathtt{X}_{pending} \cup (\mathtt{A}, op))$
$(\forall\ \mathtt{A} \in \vartheta(\mathtt{X}_{waiting} - \mathtt{X}_{sleeping}))(\mathtt{X}_{waiting} = \mathtt{X}_{waiting} - (\mathtt{A}, op))$
$(\forall\ \mathtt{A} \in \vartheta(\mathtt{X}_{waiting} - \mathtt{X}_{sleeping}))(\mathtt{A}_{twait} = \perp)$

$$\tau_e^{2PL}(c) = \frac{(n-c)\cdot\tau_e + c\cdot(\tau_e + \frac{\tau_e}{2})}{n} \tag{3}$$

being n the number of total transactions. In fact, we suppose that the arrival time of a conflicting transaction occurs in half of execution time of the previous one. Note that no multiple conflicts are considered.

In our model we take into account both the number of transaction conflicts c and the number of not compatible transaction operations i. We can model the probability of having k not compatible conflicts among transactions as:

$$P(k) = \frac{(C_{i,k}\cdot C_{n\text{-}i,c\text{-}k})}{C_{n,c}} \tag{4}$$

being $C_{z,m} = \binom{z}{m}, z \geq m, 0$ if $z \geq m$.

Considering such probability, the execution time (which behavior is reported in Fig. 1 by fixing $\tau_e = 1$) in our approach is in the hypothesis if instantaneous execution of each system secure transaction:

$$\tau_e^{our}(c,i) = \sum_{K=0}^{min(i,c)} P(k)\cdot\tau_e^{2PL} \tag{5}$$

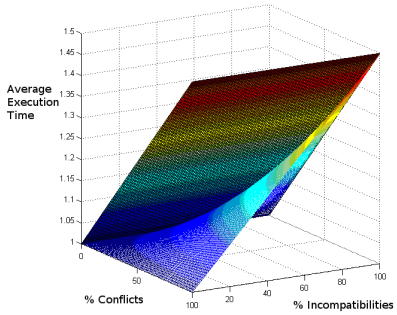

Fig. 1. Average transaction execution time

It is possible to observe that the 2PL execution time does not depend on the number of not compatible operations and grows in a linear manner as respect to the number of conflicts. In our approach, we observe an increase of times as respect to the number of transaction conflicts and the number of not compatible transaction operations. However the times are lower than 2PL ones because we do not take into account the overhead due to the reconciliation operations and *SST* execution.

Our method is more suitable as respect to 2PL when we have medium-high percentage of conflicts and medium-low percentage of incompatibilities. In our best case ($c = 100\%, i = 0$) the proposed approach presents a theoretical time improvement of 50% respect to 2PL: such enhancement is significant for high values of τ_e, in other terms for long running transactions.

In the opposite the abort percentage of sleeping transactions has been analyzed at the variation of: (i) number of transaction conflicts, (i) number of not compatible transaction operations, (iii) percentage of sleeping transactions. In the 2PL approach we can simply consider the abort percentage as function of sleeping timeout. Instead, in our approach such percentage can be computed by product of the probabilities (percentage) of having a sleep (e.g. due to a disconnection) $P(d)$, a conflict $P(c)$ and an incompatibility $P(i)$: $P(Abort) = P(d)\cdot P(c)\cdot P(i)$.

In Fig. 2 the abort percentage is reported in function of the percentage of transaction conflicts and the percentage of disconnected transactions for increasing value of the number of not compatible transaction operations.

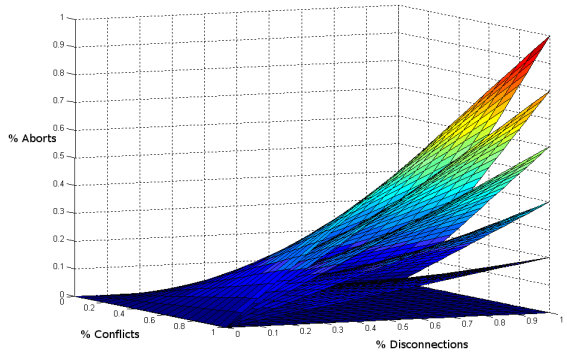

Fig. 2. Abort percentage of disconnected/sleeping transactions

B. GTM performances

The main aim of this subsection is to quantitatively evaluate the performances of our GTM (implemented in Phyton technology) in terms of transaction execution time and abort percentage.

Starting from a data set constituted by 1000 transactions that perform a subtraction (e.g. clients with a mobile device that book a flight ticket $\mathtt{X}_q = \mathtt{X}_q - 1$) or assignment (e.g admin with a fixed device that set the price $\mathtt{X}_p = 100$) operation on a single resource of a set of 5 database objects, we have automatically generated 15 classes of transactions considering α $(1-\alpha)$ as probability that a transaction performs a subtraction (assignment) operation, β as disconnection probability of subtraction transactions (no disconnections are considered for transactions with assignment), $\gamma_j^i(\sum_j \gamma = 1)$ as the probability that the i-th transaction works on j-th database object.

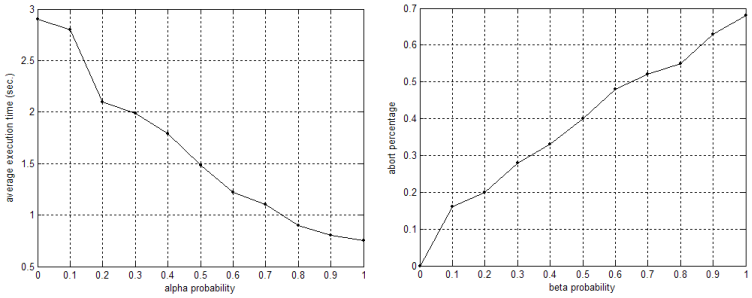

Fig. 3. GTM perfomances

Thus each class is described by: $\mathcal{C} = \langle \mathcal{T}, op, \mathcal{X}, \eta \rangle$, $\mathcal{T}$ being the set of transaction belonging to the class, *op* is the performed operation, $\mathcal{X}$ is the database object and η a boolean variable that indicates if a transaction has a disconnection. We suppose that all disconnections take place during the transaction execution and that, assigned a label $\lambda(1...1000)$ indicating the arrival order of each transaction, the interarrival time is 0.5 sec. Moreover we consider $\gamma_j^i = 10\% \forall i$.

The figure 3 reports the average execution time in sec. for transaction at α probability variation by fixing $\beta = 0.05$ and the abort percentage at β probability variation by fixing $\alpha = 0.7$.

VII. Conclusions and Future Works

In this work we have proposed a novel approach for managing concurrency of long running transactions, based on commutativity theory. We have shown that such solution presents, under certain conditions, different advantages as respect to the 2PL original protocol.

A first problem of this model could be due to possible conditions of *starvation* for incompatible transactions that try to access to resources locked by different compatible transactions. Possible solutions for this problem are: the introduction of a transaction priority or the lock-deny on a given resource for compatible transaction, if in the resource queue there are a certain number of incompatible transactions that are in a waiting state. A second problem is connected to the possibility of a *high rate of aborts* due to the violation of integrity constraints on the database, during the data reconciliation process. A possible solution for this problem is to limit the number of possible concurrent and compatible transactions on a given resource, in function of the current value X of the resource.

For what concerns deadlock, our model does not introduce any further conditions respect to 2PL protocol. Classical approaches as timeout or wait for graphs techniques can be used to detect the deadlock presence. Eventually, we also mention a last – but not least – problem. We have, in fact, assumed that *SST* is always correctly executed: further studies have to be devoted to the investigation of recovery strategies, in case of *SST* failure.

References

[1] D. Barbara', "Mobile Computing and Databases-Survey," *IEEE Trans. on Knowledge and Data Enginnering*, vol. 11, n. 1, pp. 108-117, 1999.

[2] G. Bernard et al., "Mobile Databases: a Selection of Open Issues and Research Directions," *SIGMOD Record*, vol. 33, n. 2, pp. 78-83, 2004.

[3] C. Bobineau et al., "Comparing transaction commit protocols for mobile environments," in *Proc. of IEEE Int. Conference on Database and Expert Systems Applications (DEXA)*, 2004, pp. 673-677.

[4] A. Chiu, B. Kao, and K. Lam, "Comparing Two-Phase Locking and Optimistic Concurrency Control Protocols in Multiprocessor Real-Time Databases," in *Proc. of IEEE Joint Workshop on Parallel and Distributed Real-Time Systems (WPDRTS/OORTS)*, 1997, pp. 141-148.

[5] R.A. Dirckze, and L. Gruenwald, "A pre-serialization transaction management technique for mobile multidatabases," *Mobile Networks and Applications (MONET)*, vol. 5, n. 4, pp. 311-321, 2000.

[6] U. Halici, and A. Dogac, "An optimistic locking technique for cocurrency control in distributed databases," *IEEE Transaction on Software Engineering*, vol. 17, n. 7, pp. 712-724, 1991.

[7] J. Jing, O. Bukhres, and A.K. Elmagarmid,"Distributed lock management for mobile transactions," in *Proc. of Int. Conf. on Distributed Computing Systems (ICDCS)*, 1995, 118-125.

[8] K.Y. Lam, T.W. Kuo, W.H. Tsang, and G.C.K. Law, "Concurrency control in mobile distributed real-time database systems," *Information Systems*, vol. 25, n. 4, pp. 261-286, 2000.

[9] G.Q. Liao et al., "Concurrency control of real-time transactions with disconnections in mobile computing environment," in *Proc. of ICCNMC Int. Conf.*, 2003, pp. 205-212.

[10] Q. Lu, and M. Satynarayanan, "Improving data consistency in mobile computing using isolation-only transactions," in *Proc. of IEEE HotOS Topics Workshop*, 1995, pp. 124-128.

[11] S.H. Phatak, and B.R. Badrinath, "Conflict resolution and reconciliation in disconnected databases," *Proc. of IEEE Conf. on Database and Expert Systems Applications (DEXA)*, 1999, pp. 76-81.

[12] E. Pitoura, and B. Bhargava, "Data consistency in intermittently connected distributed systems," *IEEE Trans. on Knowledge and Data Engineering*, vol. 11, n. 6, 1999.

[13] N. Prabhu, and V. Kumar, "Concurrency control in mobile database systems," in *Proc. of Conf. on Advanced Information Networking and Applications (AINA)*, 2004, vol. 2, pp. 83-86.

[14] H.Schuldt et al., "Atomicity and isolation for transactional processes," *ACM Trans. Database Sys.*, vol. 27, n.1, pp. 63-116, 2002.

[15] R. Vingralek, H. Hasse-Ye, Y. Breitbart, and H. Schek, "Unifying Concurrency Control and Recovery of Transactions with Semantically Rich Operations," *Theor. Comput. Sci.*, vo. 190, n.2, pp. 363-396, 1998.

[16] G. Vossen, and G. Weikum, *Transactional information systems*, Ed. Morgan Kaufmann, 2002.

[17] G.D. Walborn, and P.K. Chrysanthis, "Supporting semantics-based transaction processing in mobile database applications," in *Proc. of Symposium on Reliable Distributed Systems (SRDS)*, 1995, pp. 31.

[18] W.E. Weihl, "Commutativity-based concurrency control for abstract data types," *IEEE Trans. on Computers*, vol. 37, n. 2, pp. 1488-1505, 1988.

[19] Li Xiao-Rong, and Shi Bai-Le, "A Prediction-Based Approach to Concurrency Control Resulting in Low Blocking Rate under High-Quality Mobile Environment," *Proocedings of CIT Conference*, 2005, pp. 502-506.