Transaction Processing in Mobile Computing using Semantic Properties

Fritz Laux
Fakultät Informatik
Reutlingen University
D-72762 Reutlingen, Germany
fritz.laux@reutlingen-university.de

Tim Lessner
School of Computing
University of the West of Scotland
Paisley PA1 2BE, UK
timlessner@lesshome.net

Abstract

Transaction processing is of growing importance for mobile computing. Booking tickets, flight reservation, banking, ePayment, and booking holiday arrangements are just a few examples for mobile transactions. Due to temporarily disconnected situations the synchronisation and consistent transaction processing are key issues. Serializability is a too strong criteria for correctness when the semantics of a transaction is known. We introduce a transaction model that allows higher concurrency for a certain class of transactions defined by its semantic. The transaction results are "escrow serializable" and the synchronisation mechanism is non-blocking. Experimental implementation showed higher concurrency, transaction throughput, and less resources used than common locking or optimistic protocols.

1. Introduction

Mobile applications enable users to execute business transactions while being on the move. It is essential that online transaction processing will not be hindered by the limited processing capabilities of mobile devices and the low speed communication. In addition, transactions should not be blocked by temporarily disconnected situations. Traditional transaction systems in LANs rely on high speed communication and trained personnel so that data locking has proved to be an efficient mechanism to achieve serializability.

In the case of mobile computing neither connection quality or speed is guarantied nor professional users may be assumed. This means that a transaction will hold its resources for a longer time, causing other conflicting transactions to wait longer for these data. If a component fails, it is possible that the transaction blocks (is left in a state where neither a rollback nor a completion is possible).

The usual way to avoid blocking of transactions is to use optimistic concurrency protocols.

In situations of high transaction volume the risk of aborted transaction rises and the restarted transaction add further load to the database system. Also this vulnerability could be exploited for denial of service attacks.

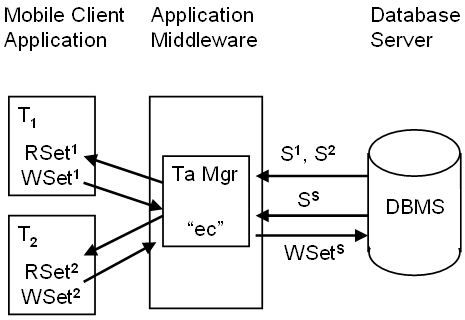

Figure 1. Three tier architecture for mobile transaction processing

In order to make mobile transaction processing reliable and efficient a transaction management is needed that does not only avoid the drawbacks outlined above but also fits well into established or emerging technologies like EJB, ADO, SDO. Such technologies enable weakly coupled or disconnected computing promoting Service Oriented Architectures (SOA).

These data access technologies basically provide abstract data structures (objects, data sets, data graphs) that encapsulate and decouple from the database and adapt to the programming models. We propose a transaction mechanism that should be implemented in the middle tier between database and (mobile) client application. This enables to move some application logic from the client to the application server (middle tier) in order to relief the client from processing and storage needs. Validation, eventual transaction rewrites, reconciliations or compensations are implemented in the middle tier as shown in Figure 1. A client transaction T_1 executes entirely locally after loading the read set $RSet^1$ into the client. On commit the middleware has to check $RSet^1$ for possible changes which happened in the mean time due to other transactions e.g. T_2 using the write set $WSet^2$. In case of serialization conflicts the transaction manager has to resolve the situation. If there are legacy applications not running through this middleware the consolidation must take into account the current database state S^s as well.

1.1. Motivation

The main differences between mobile computing and stationary computing are temporary loss of communication and low communication bandwidth. However increased local autonomy is required at the same time. Data hoarding and local processing capability are the usual answers to achieve local autonomy. The next challenge is then the synchronisation or reintegration of data after processing [1], [9], [20]. As pointed out above, blocking of host data is not an option.

The challenge is to find a non-blocking concurrency mechanism that works well in disconnected situations and that is not leading to unnecessary transaction cancellations.

We need a mechanism to reconcile conflicting changes on the host database such that the result is still considered correct. This is possible if the transaction semantic is known to the transaction management. In this paper we propose to automatically replay the transactions in case of a conflict.

We illustrate the idea by an example and defer the formal definition to the next Section. Assume that we have transactions T_1 and T_2 that withdraw €100 and €200 respectively from account a. If both transactions start reading the same value for a (say €1000) and then attempt to write back $a :=$ €900 for T_1 and $a :=$ €800 for T_2 then a serialization conflict arises for the second transaction because the final result would lead to a lost update of the first transaction.

However, if in this case the transaction manager aborts the transaction, re-reads a (= €900 now) and does the update on the basis of this new value then the result (= €700) would be considered as correct. In fact, it resulted in a serial execution from the host's view. Clearly this transaction replay is only allowed if it is known that the second transaction's subtract value does not depend on the account value (balance). This precondition holds within certain limits for an important class of transactions: Booking tickets, reserving seats in a flight, bank transfers, stock management.

There are often additional constraints to obey: A bank account balance must not exceed the credit limit, the amount on stock cannot be negative, etc.

We will introduce a transaction model based on this idea that allows higher concurrency for a certain class of transactions defined by its semantic. The transaction results are "escrow serializable" and the synchronisation mechanism is non-blocking.

1.2. Related Work

For making transaction aborts as rare as possible essentially three approaches have been proposed:

- Use the semantic knowledge about a transaction to classify transactions that are compatible to interleave.
- Divide a transaction into subtransactions.
- Reconcile the database by rewriting the transaction in case of a conflict.

Semantic knowledge of a transaction allows non serializable schedules that produce consistent results. Garcia-Molina [2] classifies transactions into different types. Each transaction type is divided into atomic steps with compatibility sets according to its semantic. Transaction types that are not in the compatibility set are considered incompatible and are not allowed to interleave at all. Farrag and Özsu [3] refine this method allowing certain interleaving for incompatible types and assuming fewer restrictions for compatibility. The burden with this concept is to find the compatibility sets for each transaction step which is a $\mathcal{O}(n^2)$ problem.

Dividing transactions into subtransactions that are delimited by breakpoints does not reduce the number of conflicts for the same schedule but a partial rollback (rollback to a subtransaction) may be sufficient to resolve the conflict. Huang and Huang [4] use semantic based subtransactions and a compatibility matrix to achieve better concurrency for mobile database environments. Local autonomy of the clients may subvert the global serializability. The solutions proposed by Georgakopoulous et al. [5] and Mehrotra et al [6] came for the prize of low concurrency and low performance. Huang, Kwan, and Li [7] achieved better concurrency by using a mixture of locking to ensure global ordering and a refined compatibility matrix based on semantic subtransactions. Their transaction mechanism still needs to be implemented in a prototype to investigate its feasibility.

The reconciliation mechanism proposed in this paper attempts to replay the conflicting transactions and produce a serializable result. This method has been investigated in the context of multiversion databases. Graham and Barker [8] analysed the transactions that produced conflicting versions. Phatak and Nath [9] use a multiversion reconciliation algorithm based on snapshots and a conflict resolution function. The main idea is to compute a snapshot for each concurrent client transaction which is consistent in terms of isolation and leads to a least cost reconciliation. The standard conflict resolution function integrates transactions only if the read set $RSet$ of the transaction is a subset of the snapshot version $S(in)$ into which the result needs to be integrated. In the case of write-write conflicts this is not the case, as $RSet \not\subseteq S(in)$.

We illustrate this by an example using the read-write model with Herbrand semantics (see [10]). Assume we have two transaction: $T_1 = (r_1(a), w_1(a), r_1(b), w_1(b))$ transfers €100 from account a to account b and $T_2 = (r_2(b), w_2(b))$ withdraws €100 from account b. If both transactions are executed in serial, the balance for account b will end up with its starting value. Now assume, that snapshot version $V(0) = \{a^0, b^0\}$ is used and both transactions start with the same value b^0. Assume the schedule $S = (r_1(a^0), r_2(b^0), r_1(b^0), w_1(a^1), w_2(b^1), c_2, w_1(b^1), c_1)$. S is not serializable and no other schedule either if both

Table 1. Comparison of high concurrency mechanisms

Mechanism	Bibliography	Drawbacks
uses Ta semantics to build compatibility set	[2], [3]	semantic classification complexity is $\mathcal{O}(n^2)$
uses subtransactions to build compatibility matrix	[4], [5], [6] [7]	manual division into sub-Ta, $\mathcal{O}(n^2)$
uses multiversions and conflict resolution function	[8], [9]	not performant in case of hot spots
uses semantic to reconcile Ta (escrow-serializability)	[24]	semantic dependency function required

transactions use the same version of b. The last transaction attempting to write account b will produce a lost update and should abort.

The multiversion snapshot based reconciliation algorithm of Phatak and Nath [9] will not be able to reconcile T_1 as $RSet(T_1) = V(0) = \{a^0, b^0\} \not\subseteq V(1) = \{a^0, b^1\}$. $V(1)$ is the result of transaction T_2. If no snapshot would have been taken and making sure that the update (read-write sequence) of b is not interrupted (interleaved) the result would have been the serializable schedule $R = (r_1(a), w_1(a), r_2(b), w_2(b), c_2, r_1(b), w_1(b), c_1)$. This shows the limitations of snapshot isolation compared to locking in terms of transaction rollbacks. On the other hand the schedule R leads to low performance because no interleaving operations for the read-write sequence are allowed. Table 1 gives an overview on transaction mechanisms used to reduce or resolve concurrency conflicts.

Our approach is to abort a conflicting transaction and automatically replay the operation sequentially. The isolation level should be read committed to avoid cascading rollbacks or compensation transactions. To ease the reconciliation processing we classify transaction in terms of its semantic.

2. Transaction Model

A database D may be viewed as a finite set of entities $a, b, \cdots, x, y, z$ (see [11]). If there exists more than one version of an entity, we denote it with the version number, e.g. x^2. These entities will be read (read set $RSet$) and modified (write set $WSet$) by a set of transactions $\mathbf{T} = \{T_1, T_2, \cdots, T_n\}$. The database D at any given time exists in a particular state D^S. A snapshot of D is a subset of a database state D^S (see [12]).

Our mobile computing system consists of a database server, an application middleware with mobile transaction management, and a mobile client with storage and computing capabilities as sketched out in Figure 1. A mobile transaction is a distributed application that guaranties transactional properties. We assume that the data communication is handled transparently by a communication protocol that can detect and recover failures on the network level. Mobile client and server have some local autonomy so that in case of network disconnection both sites can continue their work to some extent.

The data base consists of a central data store and snapshot data (at least the $RSet$) on the mobile client for each active transaction. From a transactional concept's view the transactions on the client are executed under local autonomy. The local commit is "escrowed" along with the changes to the server. The transaction manager tries to integrate all escrowed transactions into the central data store. In case of serialization conflicts reconciliation can be achieved if the semantic of the transaction is known and all database constraints are obeyed.

2.1. Escrow Serializable

For the sake of availability we want to avoid locked transaction as far as possible. One approach is to use optimistic concurrency, the other way is to relax serializability. Optimistic concurrency suffers from transaction aborts when a serialization conflicts arises [13]. The multiversion based view maintenance could minimize that risk but it requires a reliable communication at all times [14].

Much research was invested to optimize the validation algorithms [15], [16], [17], [18] for serialization. We prefer to allow non-serializable schedules that produce consistent results for certain types of transactions similar to [19].

A transaction T transforms a consistent database state into another consistent state. This may be formalized by considering a transaction as a function operating on a subset of consistent database states $\mathbf{D}$, i.e. $D^2 = T(D^1)$ with suitable $D^1, D^2 \in \mathbf{D}$, where $RSet \subseteq D^1$ and $WSet \subseteq D^2$. If we want to make the user input u explicit we write $D^2 = T(D^1, u)$.

Definition 1: (escrow serializable) Let Q be a history of a set of (client) transactions $\mathbf{T} = \{T_1, T_2, \cdots, T_n\}$ that are executed concurrently on a database D with initial state D^0. For each transaction T_i the user input is denoted by u_i. The history Q is called *escrow serializable* (ec) if

1) there exists a serial history S for $\mathbf{T}$ with committed database states $\mathbf{D^S} = (D^1, D^2, \cdots, D^n)$, where
2) $\exists r \in \{1, 2, \cdots, n\}$ with $D^1 = T_r(D^0, u_r)$ and
3) $\exists s \in \{1, 2, \cdots, n\}$ with $D^k = T_s(D^{k-1}, u_s)$ for each $k = (2, 3, \cdots, n)$

Please note that this kind of serializability is descriptive as it is not based on the operations but on the outcome (semantic) of the transactions. Escrow serializability means that the outcome is the same as with a serial execution using the same user input.

The name *escrow serializable* stems from the idea that a mobile client "escrows" its transaction to the server. On the

server site the transaction manager reconciles the transaction if all database constraints are fulfilled. This can be achieved by analysing the conflicting transactions and producing the same result as a serial execution would have done. We demonstrate this with the following example:

Example 1: (withdraw) Let T_1 and T_2 be two withdraw transactions that lift-off €100 resp. €200 from account a. We denote by c_i (resp. a_i, ec_i) the commit (resp. abort, escrow commit) command. The history

$$S^c = r_1(a)r_2(a)w_2(a' := a - 200)ec_2w_1(a'' := a - 100)ec_1$$

normally produces a lost update, but it is escrow serializable. The transaction manager on the server will detect the conflicting transaction. T_1 is aborted and automatically replayed with the previous input data. The resulting history on the server will be

$$S^s = r_1(a)r_2(a)w_2(a' := a - 200)c_2w_1(a - 100)a_1r_1(a')w_1(a' - 100)c_1.$$

Schedule S^s is equivalent to the serial execution (T_2, T_1).

If there exists a constraint, say $a > 0$ any violating transaction has to abort. Assume that $a = 300$ and take the same operation sequence as in schedule S^c then transaction T_1 has to abort because $a' - 100 \not> 0$.

2.2. Escrow Reconciliation Algorithm

Escrow serialization relies on reconciling transaction in such a way that the outcome is serializable. This is only possible if the semantic of the transactions including the user input are known. The idea is to read all data necessary for a transaction and defer any write operation until commit time. If a serialization conflict arises at commit time this means that a concurrent transaction has already committed. In this case the transaction is aborted and automatically replayed with the same input data.

Our transaction model is divided into two phases:

- **client phase**
  During the processing on the client site, data may only be retrieved from the server. It is important that the read requests are served in an optimistic way. Technically a read set of data, a data graph or any other snapshot could be delivered to the mobile client. The client transaction terminates with an escrow commit (ec) or an abort (a).
- **server phase**
  When the server receives the ec along with the write set and no serialization conflict exists the transaction is committed. In case of a conflict the transaction is aborted. The replay is done automatically with pessimistic concurrency control or serial execution. This prevents the starvation [12] of a transaction. If no constraints are violated the replayed transaction is committed.

ensure: set of transactions $\mathbf{T} = \{T_1, T_2, \cdots, T_n\}$
ensure: actual database state D^s, set of constraints $C(D)$
ensure: only committed data in read set $RSet(i)$ of T_i
ensure: $T_i = (op_{ik}, i = 1, 2, \cdots, k_i)$
for $\forall\, ec_i \in \{ec_1, ec_2, \cdots, ec_n\}$ received **do**
  // test if T_i conflicts with D^s
  if $RSet(i) \subseteq D^s$ and $\forall\, c \in C(D)$: $(c = true)$ **then**
    commit T_i
  else // abort and replay transaction
    abort T_i
    ensure: serial execution
    for each $op_{ik} \in op(T_i)$ **do** op_{ik}
    if $\exists\, c \in C(D)$ with $(c = false)$ **then** // c violated
      abort T_i
    else
      commit T_i
  end if
end for

Figure 2. Reconciliation algorithm for ec serializability

A possible reconciliation algorithm using the abort-replay mechanism is presented in Figure 2.

Care has to be taken with transactions not using the abort-replay mechanism. In this case the database should work in isolation level "serializable".

If the abort-replay mechanism is always used to integrate the transactions on the server there is no need for a certain isolation level as the read sets only contain consistent results. Any competing transactions will not alter the database until the server integrates the result. As the transaction results are integrated one-by-one, no read phenomena may occur and serial results are ensured.

We describe an implementation using SDO technology later in Section 4.

So far we have illustrated the model with transactions that produce a constant change for a data item (see Example 1). The model is valid for any transaction with a known semantic (see Theorem 1). For instance the transaction $T_3 = (r_3(x), w_3(x := 1.1x), ec_3)$ increases the prize x of a product by 10%. If the first read of x and the reread differ $(r'_3(x) \neq r_3(x))$, then the replay will produce a 10% increase based on the actual value.

For an automatic replay it is is essential to know which transactions are "immune" or depend in a predicted manner from the read set. These are the candidates for escrow serializability.

There is a technical issue for the banking example. Here we do not really need the actual withdraw amount of the transaction to replay it. It is sufficient to know three database states since the new value can be calculated by $a := a^1 +$

$a^c - a^0$ where a^1 is the actual balance, a^c is the new balance calculated by the client transaction, and a^0 is the basis on which the value a^c was computed. This observation gives reason to find classes of transactions that are ec serializable without knowing the actual user input.

3. Semantic Classification of Transactions

To facilitate the task for the reconciliation algorithm we shall classify the client transaction according to their semantic, in particular the dependency of the input from the read set.

Definition 2: (dependency function) Let T be a transaction with $RSet = \{x_1, x_2, \cdots, x_n\}$ and $WSet = \{y_1, y_2, \cdots, y_m\}$ on a database D. The function $f_i : \vec{x} \rightarrow y_i$ with $\vec{x} = (x_1, x_2, \cdots, x_n)$ and $y_i \in WSet$ is called *dependency function* of y_i.
Let $x_k \in RSet$ and $y_i \in WSet$ be numeric data types for all k. If f_i is a linear function then y_i is called *linear dependent* and we can write

$$y_i = f_i(\vec{x}) = \vec{a_i}^T \vec{x} + c_i \quad (i = 1, 2, \cdots, m) \qquad (1)$$

with $\vec{a_i}^T$ being the transposed vector $a_i = (a_{i1}, a_{i2}, \cdots, a_{in})$.
If all functions f_i are linear dependent, then

$$\vec{y} = A\vec{x} + \vec{c} \qquad (2)$$

with $m \times n$-Matrix $A = (a_{ik})$ and m-dimensional vector $\vec{c}$. We call the corresponding transaction T *linear dependent*.
If $f_i(\vec{x}) = \vec{1}^T \vec{x} + c_i$ then f_i is called *linear dependent with gradient 1* ($\vec{1}$ is the vector with magnitude 1).
If $f_i(\vec{x}) = \vec{b}^T \vec{x} + c_i$ then f_i is called *linear dependent with gradient* $\vec{b}$.

In our banking example the accounts are linearly dependent with gradient 1. The 10% price increase is an example for a transaction that is linearly dependent with gradient $b = 1.1$.

If the values of the $WSet$ however depend in an non-formalized user dependent manner from the $RSet$ then there is no way to reconcile the transaction automatically. The escrow serializable execution of a transaction depends on the fact that the outcome does change in a known functional manner.

Theorem 1: Let $\mathbf{T}$ be a set of transactions where each transaction T has known dependency functions f_i (i = 1,2, ..., m). Then the concurrent execution of $\mathbf{T}$ is escrow serializable using the abort-replay algorithm of Figure 2.

Proof: Let D^0 be a consistent state of a database with transactions $\mathbf{T} = \{T_1, T_2, \cdots, T_n\}$. Let H be a history of $\mathbf{T}$ and let w.l.o.g. the commit order be the same as the transaction index. We construct a serial transaction order that matches the definitions of escrow serializability using the abort-replay algorithm. Any write operations of the transactions T_i are postponed until commit time. The read set of T_1 is a subset of database state D^0. Then we have $D^1 = D^0 \cup S^1 := T_1(D^0)$ after the first commit c_1. When a subsequent transaction T_k tries to commit and $RSet_k \cap (\cup_{\kappa \leq k-1} S^\kappa) = \emptyset$ then there is no serialization conflict and the commit succeeds. In case of a conflict, the transaction is aborted and replayed with the same user data. During the replay the algorithm ensures serial execution, so further commits are queued. Finally we have $D^k = D^{k-1} \cup T_s(D^{k-1}, u_s)$ for $k = 1, 2, \cdots, n$. QED

Let $\{r_1, r_2, \cdots, r_n\} \subseteq D^c$ be the read set values of a client transaction and let $\{s_1, s_2, \cdots, s_n\} \subseteq D^s$ be the read set values on the server when the transaction tries to commit. Then the abort-replay mechanism produces $WSet(T) = T(D^s, \vec{u}) = A\vec{s} + \vec{u}$. The value of any numerical data item $x \in WSet$ for a linear dependent transaction is computed as

$$\begin{aligned} x^T &= \Pi_x T(D^s, \vec{u}) = \vec{a}^T \vec{s} + u \\ &= \vec{a}^T \vec{s} + (\vec{a}^T \vec{r} + u) - \vec{a}^T \vec{r} \\ &= \vec{a}^T (\vec{s} - \vec{r}) + \Pi_x D^c \\ &= \Pi_x A(\vec{s} - \vec{r}) + \Pi_x D^c \end{aligned} \qquad (3)$$

From the above equation we see that the reconciliation for transactions with a linear dependent write set may be simplified. For the transaction manager it is sufficient to know the client state D^c, the read set D^s at commit time and the state produced by $T(D^c, u)$.

Corollary: A linear dependent transaction can be reconciled (replayed) in a generic way, if client state D^c at begin of transaction, the read set D^s at commit time and the state produced by $T(D^c, u)$ are known.

The corollary statement is similar to the reconciliation proposed by Holliday, Agrawal, and El Abbadi [20].

3.1. Quota Transaction

In many cases the semantic of a transaction has well known restrictions. We can guaranty the successful execution of certain transactions if the user input remains within a certain value range.

Assume a reservation transaction. If the transaction is given a quota of q reservations then the success can be guarantied for reservations within these limits. It is the responsibility of the transaction manager to ensure that the quota does not violate the consistency constraints. For example if there are 10 tickets left and the quota is set for

2 tickets, then only 5 concurrent transactions are allowed. As soon as a transaction terminates with less than two reservations the transaction manager may allow another transaction to start with a quota that ensures no overbooking.

Quota transactions in this sense are similar to increment or decrement of counter transactions with escrow locking (see [10]).

Definition 3: (quota transaction) Let T be a transaction with $WSet = \{y_1, y_2, \cdots, y_m\}$ on a database D. For each y_i there is a value range $I := [l, u]$ associated. T is called *quota transaction* if the success of the transaction can be guarantied in advance if the result values y_i do not exceed the quota, i.e. $y_{i(old)} + l \leq y_{i(new)} \leq y_{i(old)} + u$.

Setting quotas is a mean to guaranty success for a transaction by reserving sufficient resources without locking the resources. Caution has to be taken when using quotas as resources are reserved that finally should be taken or given back. Therefore a time out or a cancel operation is required on the server site.

4. Example Implementation of the ec Model with SDO

Service Data Objects (SDO [21], [22]) are a platform neutral specification and disconnected programming model, which enables dynamic creation, access, introspection, and manipulation of business objects.

Our implementation (see Lessner [23]) of the transaction manager (TM) uses SDO graphs and resides between the data access service (DAS) and the client. This way, the TM fits well into SDO's vision of being independent of the data source.

A snapshot of each delivered graph is taken by the TM and each SDO graph associates a change summary that complies with the requirements for optimistic concurrency control (see Section 2). To assert "escrow serializabilty" (provided by the reconciliation algorithm) an association between a transaction and the semantic of this transaction is needed[1]. This association results in a classification of the transaction (e.g. "linear dependent").

An association between a classification and a verifier (see Figure 3) enables a semantic transaction level (e.g. quota verifier, escrow concurrency (EC)).

Assume that we have an incoming transaction (T-Level=EC) with a changed data graph. The transaction handler delivers the transaction to the verifier. To ensure EC, optimistic concurrency control (OCC) is checked first. If OCC is passed there is no serialization conflict. In case of a conflict the semantic level concurrency control (T-Level=EC) is invoked. This means, two verifier implementations come into play (Step 1, Figure 4) in the order of decreasing rigor (OCC → EC).

1. In heterogeneous environments an additional data object could be used to describe the semantic of a transaction, SDO uses XML as protocol

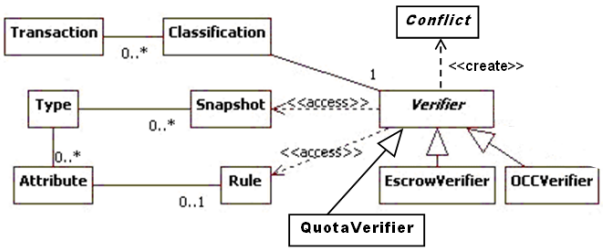

Figure 3. Abstract design of the transaction manager

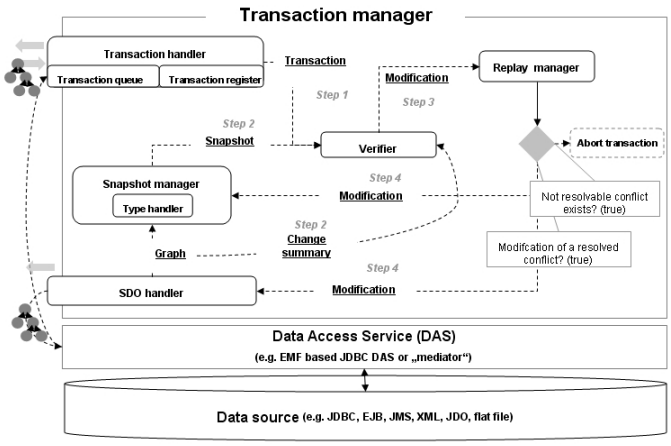

Figure 4. Architecture of the transaction manager

Each changed attribute is OCC validated against the snapshot (Step 2, Figure 4). If an OCC conflict exists and the attribute associates a known "dependency function", then reconciliation is possible and a conflict object is instantiated that represents the transaction rewrite (withdraw correction). If an OCC conflict occurs for an attribute that is not corrigible, the transaction has to abort. In a second step the EC verifier tries to resolve the conflicts (e.g. to reread the balance from the latest snapshot). If any conflicts exist after the EC verifier has finished (e.g. the withdraw amount would exceed the credit limit), the transaction has to abort, too. Each time a conflict is eventually resolvable the modification is sent to the replay manager (Step 3, Figure 4) who handles the snapshot's modifications and the graph's changes (Step 4, Figure 4). Finally all changes are written and committed to the database. The database requires isolation level "repeatable read" during steps 2 to 4.

The snapshot is data centric, which means that there exists a snapshot version of each delivered data object related to a transaction. Therefore the knowledge about the type's schema is necessary. To acquire this knowledge we decided to implement a separate meta schema. Another possibility would be, to use the schema provided by the implementation

of the SDO Data Access Services (DAS). The first possibility fits better into a general usage of the TM but causes schema redundancy. In both cases a type handler module is needed, either for accessing the schema of the SDO DAS or for accessing the additional schema.

We ran a series of simulations of concurrent withdraw transactions accessing the same account. The transaction configuration parameters were as following: Reading the balance took less than 10 ms, the user's thinking time was randomly chosen between 1 and 2 seconds, and the write time needed about 10 ms. The throughput results for up to 30 concurrent transactions is shown in Figure 5.

Running 30 transactions in parallel generated 23 serialization conflicts which triggered the replay mechanism. The net processing time for a transaction or a replay was approximately 20 ms. The total elapsed time for all 30 transactions was t = 2.1 sec which is consistent with the minimum thinking time (1 sec) plus the time for processing 30 transactions ($(30 + 23) \times 20$ ms = 1.06 sec) in *escrow*-serialization mode. The results show that we achieved an elapsed time close to the theoretical limit considering the number of replays necessary.

The nearly linear growth of the throughput when using the *escrow* concurrency control indicates that we have not reached the throughput limit. Given a processing time of 20 ms, the theoretical limit for this scenario ("hot spot" on the balance) would reach 50 transactions per second.

In contrast, the traditional OCC and locking schemes could not interleave the transactions and resulted in essentially serial processing. Therefore the performance saturated at $1/1.5 = 0.66$ transactions per second, where 1.5 sec are the average transaction duration.

In order to have a more complex example than the withdraw transaction we used the popular TPC-C benchmark [25] and analysed the *New_Order* (NOrder-Ta) and the *Payment* (Pay-Ta) transactions. The NOrder-Ta exhibits two "hot spots" with linear dependency semantics defined in section 3. One is the update of the next order id (d_next_o_id) in the district table and the other is the update of the quantity on stock (s_quantity) for each line item. The Pay-Ta contains "hot spots" in the tables *Warehouse, District*, and *Customer*. Again, the semantics of the transaction is linear dependent with gradient 1 (see section 3) as it deals with updating three balances with a fixed amount, updating the year to date payment by the same amount, and incrementing the payment count.

In total we have identified 7 situations where the *escrow*-serialization mechanism could be beneficial for performance. First tests indicate a substantial improvement over traditional locking mechanism.

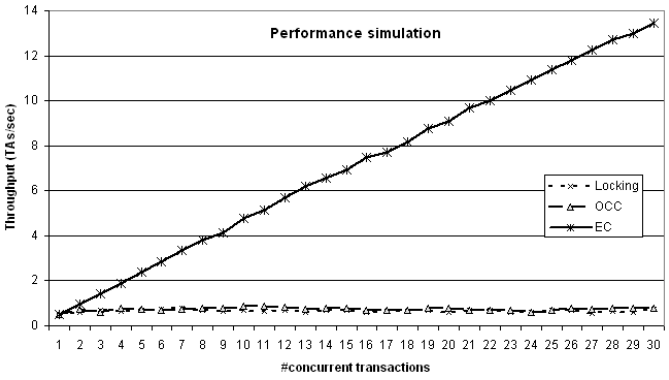

Figure 5. Performance of EC compared with OCC and locking

5. Conclusions

Mobile transactions have special demands for the transaction management. We propose a transaction model that is non-blocking and is reconciling conflicting transactions by exploiting the semantic of the transaction. A simple abort-replay mechanism can produce reconciliation in the sense of escrow serializability. The abort-replay algorithm detects conflicts by rereading the data. The mechanism is easy to implement and can make use of update operation when read - and write set overlaps.

If all writes are postponed until the commit is issued and the reread and write operations during reconciliation are executed serialized or serial, then no inconsistent data will be read. A further option is to use consistent snapshots. Independent from the mechanism the read phase should be executed with optimistic concurrency control.

In contrast, the reconciliation phase should run in a preclaiming locking mode. This ensures efficient sequential processing of competing transactions without delays as user input is already available and starvation is avoided. With this marginal condition the escrow serialization algorithm has the potential to outperform other mechanisms.

For the class of linear dependent transactions it is sufficient for reconciliation to know the client state at begin of transaction, the state produced by the client transaction on the client site, and the database server state at commit time.

Acknowledgements

This paper was inspired by discussions with the members of the DBTech network and the ideas presented during the DBTech Pro workshops. The DBTech Project was supported by the Leonardo da Vinci programme during the years 2002 - 2005. The present paper is a revised and extended version of [24].

References

[1] J.H. Abawajy, M. Mat Deris; *Supporting Disconnected Operations in Mobile Computing*, The 4^{th} ACS/IEEE International Conference on Computer Systems and Applications (AICCSA-06) , 911–918 (2006), American University of Sharjah, UAE.

[2] Hector Garcia-Molina; *Using Semantic Knowledge for Transaction Processing in a Distributed Database*, ACM Transactions on Database Systems, Vol. 8, No. 2 (1983), 186–213.

[3] Abdel Aziz Farrag and M. Tamer Özsu; *Using Semantic Knowledge of Transactions to Increase Concurrency*, ACM Transactions on Database Systems, Vol. 14, No. 4, (1989), 503–525.

[4] Shi-Ming Huang and Chien-Ming Huang; *A semantic-based transaction model for active heterogeneous database systems*, IEEE Conference on Systems, Man, and Cybernetics **3** (1998), 2854–2859

[5] D. Georgakopoulous, M. Rusinkiewicz, and A. Sheth; *On Serializability of Multi-database Transactions through Forced Local Conflicts*, Proceedings of the 7^{th} Conference Data Engineering, IEEE publication (1991), 314–323

[6] S. Mehrotra et al; *Non-serializability Executions in Heterogeneous Distributed Database Systems*, Proceedings on the 1^{st} International Conference Parallel and Distributed Information Systems, IEEE publication (1991), 245–252

[7] Shi-Ming Huang, Irene Kwan and Chih-He Li; *A Study on the Management of Semantic Transaction for Efficient Data Retrieval*, SIGMOD Record **31** (3) (2002), 28–33

[8] Peter C. J. Graham, Ken Barker; *Effective Optimistic Concurrency Control in Multiversion Object Bases*, Object-Oriented Methodologies and Systems, International Symposium ISOOMS '94, Palermo, Italy (1994), 313-328, Lecture Notes in Computer Science, Springer

[9] Shirish Hemant Phatak and Badri Nath; *Transaction-Centric Reconciliation in Disconnected Client-Server Databases*, Mobile Networks and Applications **9** (2004), 459–471, Kluwer Academic Publisher, The Netherlands.

[10] G. Weikum, G. Vossen; *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*, Morgan Kaufmann Publishers, San Francisco, CA, 2002

[11] Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom; *Database Systems: The Complete Book*, Prentice Hall, New Jersey, 2002.

[12] Avi Silberschatz, Hank Korth, S. Sudarshan; *Database System Concepts,* 5^{th} *ed.*, McGraw Hill, New York, NY, 2006.

[13] SongTing Chen, Bin Liu, and Elke A. Rundensteiner; *Multiversion-Based View Maintenance Over Distributed Data Sources*, ACM Transactions on Database Systems, Vol. 29, No. 4 (2004), 675–709.

[14] JuhnYong Lee; *Precise serialization for optimistic concurrency control*, Data & Knowledge Engineering, Vol. 29, Issue 2 (1999), 163–179, Elsevier Science B.V., The Netherlands.

[15] K. A. Momin, K. Vidyasankar: *Flexible Integration of Optimistic and Pessimistic Concurrency Control in Mobile Environments*. ADBIS-DASFAA, 346–353 (2000), Prague, Czech Republic

[16] Stefano Ceri, Susan S. Owicki; *On the Use of Optimistic Methods for Concurrency Control in Distributed Databases*, Berkeley Workshop (1982) 117-129

[17] Ho-Jin Choi, Byeong-Soo Jeong; *A Timestamp-Based Optimistic Concurrency Control for Handling Mobile Transactions*. ICCSA (**2**) 2006, 796-805

[18] Adeniyi A. Akintola, G. Adesola Aderounmu, A. U. Osakwe, Michael O. Adigun; *Performance Modeling of an Enhanced Optimistic Locking Architecture for Concurrency Control in a Distributed Database System*, Journal of Research and Practice in Information Technology **37**(4): (2005)

[19] D. Agrawal, J.L. Bruno, A. El Abbadi, V. Krishnaswamy; *Relative Serializability: An Approach for Relaxing the Atomicy of Transactions*, SIGMOD/PODS 94 (1994), Minneapolis, Minnesota, USA.

[20] Joanne Holliday, Divyakant Agrawal, Amr El Abbadi; *Disconnection Modes for Mobile Databases*, Wireless Networks **9** (2002), 391–402, Kluwer Academic Publisher, The Netherlands.

[21] M. Adams, C. Andrei et al; *Service Data Objects For Java Specification*, BEA Systems, Cape Clear Software, IBM, Oracle, et al, (2006), http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf

[22] J. Beatty, S. Brodsky, M. Nally, R. Patel; *Next-Generation Data Programming*, BEA Systems, IBM, (2003), http://www.bea.com/dev2dev/assets/sdo/Next-Gen-Data-Programming-Whitepaper.pdf

[23] Tim Lessner; *Transaktionsverarbeitung in disconnected Architekturen am Beispiel von Service Data Objects (SDO) und prototypische Implementierung eines Transaktionsframeworks*, Diploma Thesis (2007), Department of Business Informatics, Reutlingen University, Germany

[24] F. Laux, T. Lessner, M. Laiho; *Semantic Transaction Processing in Mobile Computing*, Techniques and Applications for Mobile Commerce - Proceedings of TAMoCo, Frontiers in Artificial Intelligence and Applications Vol. **169** (2008) 153 - 164, IOS Press, Amsterdam, The Netherlands

[25] N.N., *TPC Benchmark C, Standard Specification, Revision 5.9*, Transaction Processing Performance Council (TPC), (2007), http://www.tpc.org/tpcc/