Escrow Serializability and Reconciliation 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. The model copes with many mobile scenarios and is able to improve existing synchronization approaches through an automatic replay approach, whereas transaction migration or transactional composition in mobile interaction is not considered. Rather we provide an optimistic transaction model residing at middleware layer. 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. A reliable end-to-end protocol (ISO/OSI level 4) like TCP is not sufficient as a user transaction (ISO/OSI level 7) may span multiple sessions. The communication delay due to retransmissions occupies resources e.g blocks data elements. 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.

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.

The present paper is an extended version of [1], and it provides more detailled information about the server phase, the requirements for transaction splitting, and other

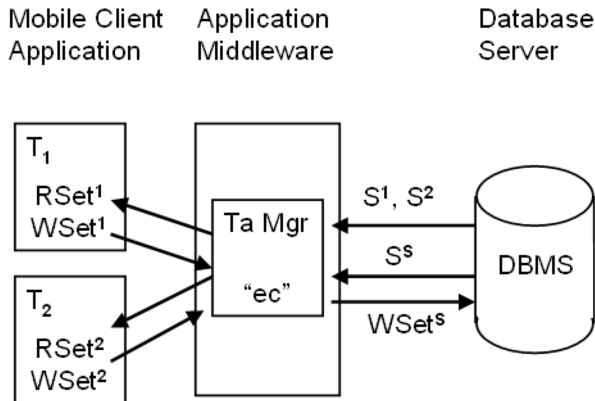

Figure 1. Three tier architecture for mobile transaction processing

implementation issues.

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 [2], [3], [4]. 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 quantity 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.

The next section sketches out the related work and addresses some drawbacks of existing approaches. Section 3 and 4 introduce our model and provide the required definitions for escrow serializability as well as the transactions' semantics. Section 5 describes theoretically the client and server phase in detail, whereas section 6 provides information about an implementation based on Service Data Objects (SDO). Section 7 focuses on an alternative conflict detection using Row Version Verification (RVV) and the performance of the escrow model is presented in section 8. The paper's conclusions are presented in section 9.

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 [5] 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 [6] 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. Our proposed model is a $\mathcal{O}(n)$ problem, because for each operation of a transaction it has to be decided if the operation is reconcilable or not, and it is not required to define the compatibility with every concurrent transaction.

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 [7] use semantic based subtransactions and a compatibility matrix to achieve better concurrency

Table 1. Comparison of high concurrency mechanisms

Mechanism	Bibliography	Drawbacks
uses Ta semantics to build compatibility set	[5], [6]	semantic classification complexity is $\mathcal{O}(n^2)$
uses subtransactions to build compatibility matrix	[7], [8], [9] [10]	manual division into sub-Ta, $\mathcal{O}(n^2)$
uses multiversions and conflict resolution function	[11], [3]	not performant in case of hot spots
uses semantic to reconcile Ta (escrow-serializability)	[13], [1]	semantic dependency function required

for mobile database environments. Local autonomy of the clients may subvert the global serializability. The solutions proposed by Georgakopoulous et al. [8] and Mehrotra et al [9] came for the prize of low concurrency and low performance. Huang, Kwan, and Li [10] 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 [11] analysed the transactions that produced conflicting versions. Phatak and Nath [3] 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 $RSet \nsubseteq S(in)$.

We illustrate this by an example using the read-write model with Herbrand semantics (see [12]). 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 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 [3] will not be able to reconcile T_1 as $RSet(T_1) = V(0) = \{a^0, b^0\} \nsubseteq V(1) = \{a^0, b^1\}$. $V(1)$ 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.

Many mobile replication and synchronization models are introduced in literature. Some of these models could be improved by an application of the *ec* - model. The Isolation Only (IO) [14] for example, enables disconnected operations in a private workspace and distinguishes between 1st and 2nd class transactions, whereas only the 1st class is serializable with all committed transactions (SR). The 2nd class is only local serializable with other 2nd class transactions (LSR). SR is granted if a local transaction was successfully reintegrated (LSR $\rightarrow$ SR). IO defines global serializability (GSR) to be the next level of serializability, and the difference between LSR, SR, and GSR is that GSR is not testable during a transaction's execution. If a test for GSR fails, the IO model also proposes to re-execute a transaction with the current DB's state. However, the IO model doesn't define the types of conflicts where re-execution is applicable. The *ec*-model is capable to extend the IO model since our model provides a semantics based classification for conflicts and a mechanism to automatically replay conflicting transactions. Furthermore, the IO model is based on Kung's OCC model to ensure SR (see [15]), and in section 5 we present how to extend an OCC with an additional reconcile phase. So the *ec* - model is able to improve GSR and SR in the IO model. The idea to replay a transaction on a stationary DBMS is referred to as transaction oriented synchronization and is described in the Two - Tier - Replication model ([16]). Reconciliation is based on the transaction's semantics. Our model focuses on the server or middleware and the transaction's semantics has to be made available for the server only, i.e. the TM. If local transactions have to be aware of the semantics, because the *ec*-model is applied to local DBMS, a "Combat" mechanism as introduced in the Pro-Motion transaction model ([[17], [18]) is a proper solution.

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. Our model relies on the optimistic snapshot validation without critical section [19] algorithm or row version verifiying (RVV) [20], and to ease the reconciliation processing we classify transaction in terms of its semantics.

3. Transaction Model

A database D may be viewed as a finite set of entities or elements $a, b, \cdots, x, y, z$ (see [21]). 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 [22]).

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.

3.1. 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 [22]. The multiversion based view maintenance could minimize that risk but it requires a reliable communication at all times [3].

Much research was invested to optimize the validation algorithms [23], [24], [25], [26] for serialization. We prefer to allow non-serializable schedules that produce consistent results for certain types of transactions similar to [27].

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 takes €100 resp. €200 from account x. We denote by c_i (resp. a_i, ec_i) the commit (resp. abort, escrow commit) command. The history

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

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

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

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

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

3.2. 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

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

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 nested transactions conflicts or the starvation [22] of a transaction. If no constraints are violated the replayed transaction is committed.

A possible reconciliation algorithm using the abort-replay mechanism is presented in Figure 2. Section 5 describes the server phase in detail and provides an extended version of this algorithm.

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.

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.

An implementation using SDO technology is described later in Section 6.

4. 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: (escrow serializable)

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 1: (escrow serializable)

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

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

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 [4].

4.1. 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 [12]).

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.

4.2. A transaction's role

We will briefly describe the idea how to apply the role pattern on transactions. Consider the situation where a transaction has to prevail against other transactions' modifications. If validation fails (e.g. a constraint was violated) there is no chance for a transaction with a higher priority to prevail against concurrent transactions. However, if we assign an owner or master role to that transaction, the TM is able to detect the role and adapt the transaction's handling. In the case of an owner, he may write modifications of the owner transaction regardless of any other transactions, and conflicting transactions have to abort.

In general, roles are a well understood concept, but transaction models do not apply this concept directly to transactions. Instead, the concept is shifted up to application level whereas our intention is to apply a role directly to the transactions. Generally, a role is represented by a logical identifier, and a set of conditions reflect the roles' intention, whereby each condition leads to activities. The role model could be implemented based on the ECA (Event, Condition, Activity) concept of Active Database Management Systems (see [28]). The event is thrown, if the TM detects a role associated with the current transaction. The conditions are validated, and the activities are executed. In our model an activity, thrown by a role, affects the TM's behaviour which isn't data centric as the ECA is.

Security aspects may complement, add or even contradict the activities implied by the transaction's role. Care has to be taken if transactions with an identical role operate on the same data, and they run into a deadlock situation, e.g. two owner roles. If this is an unwanted state, the first role is a semantic lock for other identical roles.

5. Server phase of the *ec*-model

The sections above focus on reconcilable elements, i.e. values with a linear dependency function. But, in general a transaction consists of non-reconcilable elements (not able to get corrected), too. E.g. a customer name is non-reconcilable (provided no dependency function is found). On the other hand an account balance, as in the example above, is reconcilable. Both kinds of elements may belong to the same transaction.

Reconciliation prevents only reconcilable elements from unnecessary conflicts. Therefore, we will apply the escrow model to an optimistic concurrency control (OCC) algorithm in order to handle non reconcilable elements, too. Kung and Robinson [15] describe in their paper a general approach for an OCC algorithm with three different phases; The read, validation, and write phase. The obstacle of this approach is the critical section namely validation and write. The indivisibility of these two phases leads transactions in their read phase to also interrupt their work if other transactions are validating or writing. Unland [19] suggest a validation without critical section $VAL^{\neg CS}$. Read transactions together with currently writing transactions can validate concurrently except for a short critical section when a transaction number (counter access only) is generated and assigned. The $VAL^{\neg CS}$ is extended by an additional reconcile phase in order to comply with reconcilable elements.

The $VAL^{\neg CS}$ defines a transaction to be either in the (i) read, (ii) validation or (iii) write phase. The additional reconcile phase is explained later (see figure 3 for an example). During the read phase (i) a transaction has to validate against all transactions terminating in this phase (write). This is referred to as forward-oriented optimistc concurrency control [12].
If a transaction enters the validation phase (ii) a transaction number TNR is assigned (the only critical phase), and the transaction has to validate against all transactions with a smaller transaction number not finished validation yet.
In the write phase (iii) T_i's result is published provided the write succeeds.
The *ec*-model needs an additional reconcile phase for values necessary to reconcile. A validation order within a transaction is also indispensable, since some elements need validation only, and reconcilable elements need reconciliation (see later). Generally, OCC algorithms verify that a $RSet$ of a transaction T have no intersection with another transaction's $WSet$. The intersection is determined on an entity basis, but reconciliation is on a value basis. Recall, reconciliation means to re-exceute an operation with the current value(s). Thus, for non reconcilable elements validation on an entity basis is proper, whereas reconciliation is on a value basis. Therefore, a transaction may need two different validation strategies. Furthermore, we have reconcilable values with a constraint, e.g. an account is not allowed to fall below the credit limit.

Based on the observations so far we define the following (Definition 5).

Definition 4: (escrow transaction)

A transaction consists of 1) reconcilable elements $\vec{x} = (x1, \ldots, x_i)$ (see definition 2), 2) reconcilable elements with a constraint $\vec{cx} = (cx_1, \ldots, cx_i)$, and 3) non reconcilable elements $\vec{nx} = (nx_1, \ldots, nx_i)$.For each reconcilable value x_i and cx_i the dependency function f_i of y_i is known (see defintion 2). The userinput is denoted by u.

- (1) $(\vec{y}, \vec{cy}, \vec{ny}) = T(\vec{x}, \vec{cx}, \vec{nx}, u)$
- (2) Validation of $\vec{x}$ and $\vec{cx}$ is on an element basis, whereas $\vec{nx}$ is on a variable basis.
- (3) $RSet(T) = \{\vec{x}, \vec{cx}, \vec{nx}\}$
- (4) $WSet(T) = \{\vec{y}, \vec{cy}, \vec{ny}\}$

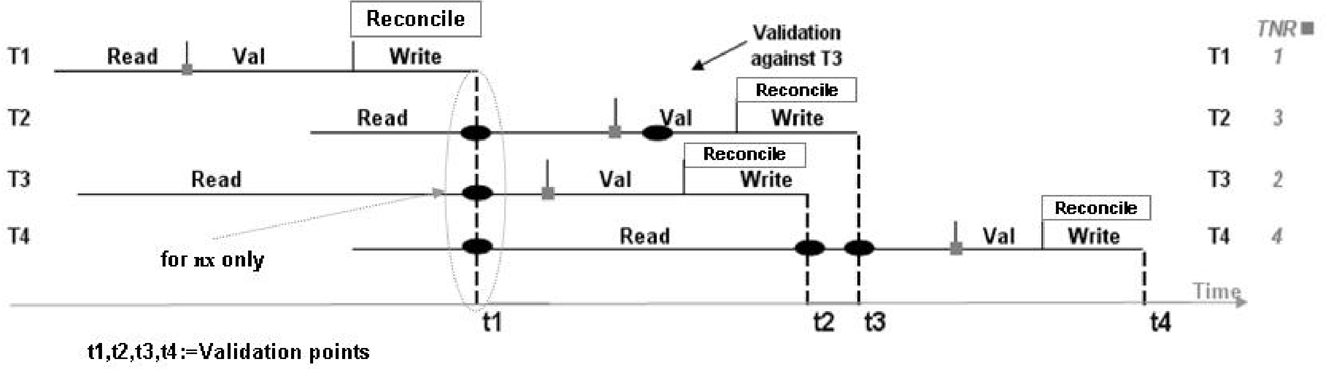

Figure 3. Modified validation without critical section ($VAL^{\neg CS}$), see [19]
$T2, 3, 4$ have to validate against $T1$ because $T1$ writes (t1). $T2$ has to validate against $T3$, because $T2$'s TNR $<$ $T3$'s TNR.

Example 2: (product inventory) Several employees receive new products. Each product is stored at exactely one storage location and employees store products concurrently. Before an employee starts to stock, he reads the product's data (id,$location$,$quantity$) via infrared using his mobile device. An infrared access point with a limited coverage resides near each product's location. After an employee finishes work he commits the changes on the mobile device, and sents his modifications ($WSet$ or change set) back to the host via infrared. Let $quantity$ be the only reconcilable value.

$T = (r(id), r(location), r(quantity), w(quantity' := quantity + a))$, where a is the amount of new products. $RSet(T) = \{\vec{x} = \emptyset, \vec{cx} = \{quantity\}, \vec{nx} = \{id; location\}\}$ (see definition 4)

The reconcilable elements depend on the transaction. In the withdraw example the user transfers money and the current balance does not affect his decision to execute the transfer, e.g. to pay a bill. However, assume the actual balance affects the decision to pay the bill. In the first situation the balance is classified as reconcilable, whereas it's non - reconcilable in the second one.

In the next section we describe the read phase's issues. The validation, and reconcile phase are described later.

5.1. Read phase

Since the $VAL^{\neg CS}$ was designed for connected architectures we analyze the read phase in order to fit the requirements for disconnected architectures and reconciliation. The support for local autonomy requires to replicate data on the mobile device. Beside this aspect, the read set contains reconcilable, as well as non-reconcilable entities (see defintion 4).

In OCC, a reading transaction is not allowed to read data which intersect with the write set $WSet$ of a transaction (see preliminaries, read phase (i)). Section 5.2 shows that reconciliation prevents from critical read anomalies. Therefore, the test in (i) reduces to $\vec{nx}$ components only. Validation of non reconcilable data is on an entity basis, so we denote $RSet^{NX}$ as the read set for non-reconcilable elements and $WSet^{NX}$ respectively. For the modified validation in the read phase of Unland's $VAL^{\neg CS}$ algorithm see figure 4.

for $\forall T_i \in \mathbf{T}^{read}$ **do**
  if $(\forall T_j \in \mathbf{T}^{write} : RSet^{NX}(i) \cap WSet^{NX}(j) \neq 0)$
  abort T_i
end for
//$\mathbf{T}^{read}$:=All reading transactions.
//$\mathbf{T}^{write}$:=All writing transactions.

Figure 4. write - read validation

5.2. Read anomalies and reconciliation

The lost update anomaly is not treated in this section because the example above (see example 1) shows that reconciliation prevents from a lost update.

Dirty read

Assume the schedule

$S = r_2(x)w_2(x' := x - 200)ec_2r_1(x')a_2w_1(x'' := x - 100)ec_1$, where ec_i indicates an escrow commit, and a_i an abort.

S normally produces a dirty read, but it is escrow serializable. The TM will detect the conflicting transaction T_1, replay it automatically, and will use the current value of x. The result is the following schedule which prevents from a dirty read:

$S = r_2(x)w_2(x' := x - 200)ec_2r_1(x')a_2w_1(x'' := x - 100)ec_1a_1r_1(x)w_1(x' := x - 100)ec_1c_1$

a_i indicates an abort by the TM, because a constraint $x > 0$ was violated, and assume x was 200 at $r_2(x)$.

Non repeatable read

Assume the following schedule:

$$S = r_1(x)r_2(x)w_2(x' := x + 200)ec_2c_2r_1(x')ec_1c_1$$

As denoted in the schedule $x \neq x'$.

The non repeatable read anomaly is predestinated to briefly sketch out the problem of read anomalies in disconnected architectures. After T_1 reads x, x is only present in T_1's workspace. Each re-read in the workspace will produce the same result and on the mobile client's side no non repeatable read is present. But the server side handling is of interest. If the operation is replayed with x's current value the read is still non-repeatable. Therefore only to lock x, or restrict repeatable read to workspace level is possible. But, the point is that the *ec*-model exploits a non-repeatable read for x values to replay the operation. Thus, the *ec*-model doesn't support an isolation of repeatable read.

Phantom read

Assume the following schedule:

$$S = cnt_{11} := count_1(X)insert_2(x)ec_2c_2cnt_{12} := count_1(X)ec_1c_1$$

Both $count(X)$ operations execute in the transaction's workspace (mobile client), and they have to produce the same result, unless a concurrent transaction accesses the same workspace (not provided by *ec*), or an explicit re - read was performed. If the $count$ operation is able to get reconciled and a phantom read should be prevented, T_1 has to be *ec*-aborted and replayed with the current state. The result is the following schedule which prevents from a phantom read:

$$S = cnt_{11} := count_1(X)insert_2(x)ec_2c_2ec_1a_1cnt_{11} := count_1(X')c_1$$

Summarizing, escrow serializability prevents reconcilable values from lost update, dirty read, but phantom read is possible.

5.3. Validation - and reconcile phase

Validation starts with an escrow commit and the associated change set. As described in the phase validate (ii), each transaction starting validation has to validate its change set against each (currently) validating transaction with a lower TNR (see figure 6).

Applying this test to transaction

$$T_w = (r(id), r(location), r(quantity), w(quantity))$$

(see example 2) means to test if $\vec{nx} = (id, location)$ or $\vec{x} = (quantity)$ intersects with another transaction's $WSet$ with a lower TNR. If validation succeeds the modifications are written, and if either $id, location$ or $quantity$ intersects with another transaction's $WSet$ the transaction aborts. But, considering that $quantity$ is a reconcilable value an intersection of $quantity$ does not lead to a unresolvable conflict provided no constraint is violated. According to that, the test in (ii) is customized to fit the requirements for reconcilable elements (see figure 6).

There are some other conclusions. For the validation of reconcilable elements it's sufficient to validate the constraint only (on a value basis). If no constraint is present, validation is unnecessary and the algorithm is directly entering the reconciliation phase, because to reconcile means just to replay the conflicting operation with the current value. Therefore, only non reconcilable entities enter the validation and write phase, whereas constrained reconcilable elements need validation and reconciliation. And in contrast, reconcilable elements enter the reconcile phase only. Recall, that reconcile includes to write data.

Assume the validation of $location$ fails. Under this circumstance the transaction aborts due to atomicity, and to prevent from unnecessary reconciliation a validation order from non-reconcile to reconcile $\vec{nx} \overset{order}{\rightarrow} \vec{x}$ has to be followed.

Following this order, to replay means that the operation which leads to a conflict is replayed only, because each write of a non-reconcilable element was still performed, and to rewrite again is unwanted due to performance reasons. Thus, the *ec*-model treats with transaction splitting, because for each element of $\vec{x}$ a new nested sub transaction is created and executed by the TM.

Conclusions:

- (1) A validation order from non-reconcilable to reconcilable prevents from unnecessary reconciliation. Denoted by $\vec{nx} \overset{order}{\rightarrow} \vec{x}$
- (2) A replay will only replay the conflicting operations.
- (3) The replay of an operation leads to a new nested transaction.

Figure 5 defines the possible states and transitions of the server phase.

Each non-reconcilable element nx_i validates first (1). If validation succeeds the value of nx_i is written (2) and committed. If validation fails, T is aborted (3).

If each nx_i was written successfully each cx_i is validated next (1). The transaction aborts if a constraint is violated (3). Provided a constraint is not violated cx_i enters reconciliation (4). In case each nx and cx commits, each x_i directly enters the reconcile phase (5). After the reconciliation and write phase a commit is only possible, since we assume a reliable and physical error free hardware (6). If hardware fails, this is a matter of recovery. Transactions, like the proposed Quota, try to prevent from constraint violation through preestimation. For such transactions a relaxed validation might be applicable. It could be indicated by a role, and classified by failure probabilities.

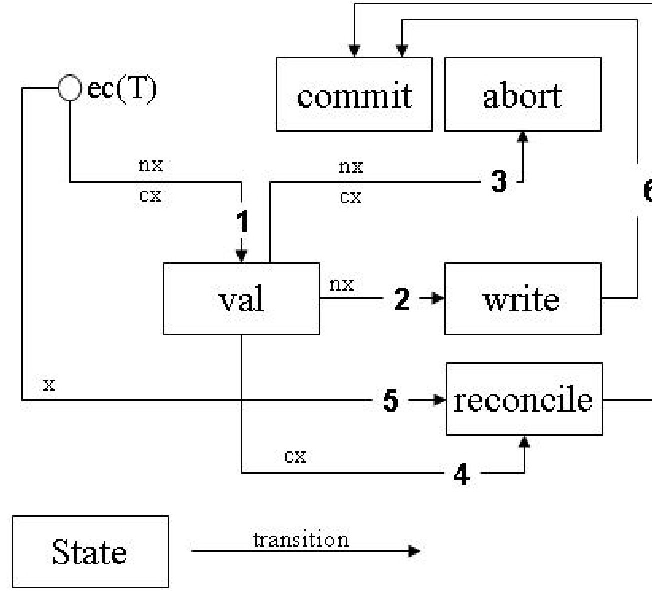

Figure 5. States of an escrow transaction

Reconciliation ensures a serial execution (see figure 6) and the performance section shows higher throughput for reconcilable transactions.

As mentioned before an escrow transaction is splitted to comply with the execution order $\vec{nx} \overset{order}{\rightarrow} \vec{x}$. Definition 5 defines how an escrow transaction is splitted by the TM.

Definition 5: (Transaction splitting)

(1) Let T' be a user transaction which spans sub transactions T_i. To ensure atomicity T' is aborted if any sub transaction $T_i \in \mathbf{T} = T_i$ fails.
(2) The set of sub transactions $\mathbf{T}$ contains one nested transaction T^{NX}, and two nested atomic sets of transactions $\mathbf{T}^{CX}$ and $\mathbf{T}^{X}$.
$T' := \{T^{NX}, \mathbf{T}^{CX}, \mathbf{T}^{X}\}$
(3) Each operation $op_n(cx) \in T'$ which modifies a **constrained** reconcilable entity leads to the creation of a new sub transaction in order to replay the operation within T'. Let $\mathbf{T}^{CX}$ represent these kind of transactions.
$\mathbf{T}^{CX} := (T_1(op_1(cx)), \ldots, T_i(op_n(cx)))$
(4) Each operation $op_n(x) \in T'$ which modifies a reconcilable entity leads to the creation of a new sub transaction in order to replay the operation within T'. Let $\mathbf{T}^{X}$ represent these kind of sub transactions.
$\mathbf{T}^{X} := (T_1(op_1(x)), \ldots, T_i(op_n(x)))$
(5) Let T^{NX} be the transaction which contains all operations $op_n(nx)$ that are non-reconcilable.
$T^{NX} := (op_1(nx), \ldots, op_n(nx))$

To split a transaction requires information about the variables and their semantics. We base our model on the idea of change sets which are delivered to the TM after local (on the mobile device) modifications took place. A change set ChS represents the user transaction T' and consists of several entities e. Beside the different versions for a variable (old and new), each e must provide a key (e.g. unique type name) to enable a mapping between transaction, reconciliation rules and constraints. Usually a transaction defines which variable is reconcilable or non - reconcilable. This information, and the type information of the change set is adequate to split a transactions into its corresponding sub transactions $T^{NX}, \mathbf{T}^{CX}, \mathbf{T}^{X}$. In our prototyp a type handler is used in order to faciliate the mapping. The reconcilable entities are defined transaction specific on a unique type basis, and after a transaction starts, it registers immediately by the TM. This registration and the type handler enable, as long as the transaction and the TM rely on an identical type basis, to split a transaction and utilize a transaction specific reconciliation.

This section described the server phase of the ec-model and how the $VAL^{\neg CS}$ algorithm is extended by an additional reconcile phase. To split a transaction is an adequate solution to handle reconcilable and non-reconcilable elements within the same transaction.

5.4. Nested transactions in the *ec*-model

So far, an user transaction is defined to be atomic, but there might be some dependencies within an user transaction

which allow to weaken the execution order. To obtain a weakened execution order we classify sub transactions according to the open nested transaction model first (see [29], [12], [30]). Generally, the nested transaction model allows to relax atomicity and isolation which is often required in mobile, transactional workflow scenarios, or other so called Advanced Transaction Models (ATM) (see [31]). In the open nested transaction model a sub transaction, or child, is:

1) open ($iso = false$), if the results are published to all transactions. It is closed ($iso = true$), if results are published to the parent transaction only (ISOLATION).
2) It is vital ($vit = true$), if to abort leads its parent to abort, too. And, non- vital ($vit = false$), if an abort does not affect its parent (Atomicity, $abort\,\overrightarrow{T_cT_p}$).
3) T depends on its parent ($dep = true$), if the parents abort leads T to abort, too; independent ($dep = false$) if not (Atomicity, $abort\,\overrightarrow{T_pT_c}$).

By definition 5 T' has to be atomic, thus each $T \in T'$ is dependent, vital, and closed in order to avoid cascading rollback (see section 2). For independent, non-vital and open children however, it's possible to change the execution order, and to execute this new class of transactions seperately (possibly pooled or even delayed). E.g. in example 2 assume an additional transaction which monitors the time an employee needs to complete an order. A $time$ variable is incremented locally and synchronized at the location with the time of all employees in order to calculate the average delivery time for that product. This is an example for an independent, non - vital and open transaction on a reconcilable value.

The drawback of intra transaction dependencies is that the TM needs to know them. Our implementation (see 6) as well as our model requires a change set reflecting local modifications. If the change set provides the information needed to classify a transaction a new execution order is applicable. Based on the classifications for sub transactions definition 6 extends definition 5.

Definition 6: (*ec*-independency)

(1) Let $\mathbf{T}^{IN}$ be the set of all independent, non-vital, and open transactions.

(2) T' is defined as an user transaction with one nested transaction T^{NX}, two nested atomic sets of transactions $\mathbf{T}^{CX}$ and $\mathbf{T}^{X}$, and one atomic, and *ec*-independent set of sub transactions $\mathbf{T}^{IN}$. Each $T \in \mathbf{T}^{IN}$ is *ec*-independent.

$T' := \{(T^{NX}, \mathbf{T}^{CX}, \mathbf{T}^{X}), \mathbf{T}^{IN}\}$

(3) A transaction T_c is *ec*-independent if the dependency function dep of T_c is known for all other transactions $T_i \in T' : T_i \neq T_c$, and for each $T_i \in T'$ dep validates to true, whereas true indicates the *ec*-independency of T_c.

(4) *ec*-independency:$iso = 0 \wedge vit = 0 \wedge dep = 0$

Although mobile transaction models deal with (non-) substitutable, compensable, temporal, and spatial transactions[1], none of them have been considered here. Such intra transaction dependencies often originate from the use case's semantics and less from the value's or operation's semantics. In an interaction scenario (e.g. workflow based) the state and transition model is able to provide the information about such intra dependencies. In mobile computing, service selection and composition is context aware and an (formal) interaction model might not given. Hahn's model [33] exploits transactional properties (non-functional) defined in the interface to determine which workflow pattern (XOR, AND, and SEQUENCE) is proper for a service interaction.

Our focus is to exploit the semantics on a data and operations level first, and not to exploit the semantics of complex interaction scenarios. The *ec*-model may provide a reliable foundation other transaction models could rest upon. To benefit from both kinds of semantics (data & value and interaction) seems to be a worthwhile objective.

6. Example Implementation of the ec Model with SDO

Service Data Objects (SDO [34], [35]) are a platform neutral specification and disconnected programming model, which enables dynamic creation, access, introspection, and manipulation of business objects.

Our implementation (see Lessner [36]) 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 3). To assert "escrow serializabilty" (provided by the reconciliation algorithm) an association between a transaction and the semantic of this transaction is needed[2]. This association results in a classification of the transaction (e.g. "linear dependent").

An association between a classification and a verifier (see Figure 8) 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 implemen-

1. Temporal and spatial describe transactions subject to temporal and spatial restrictions, respectively.

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

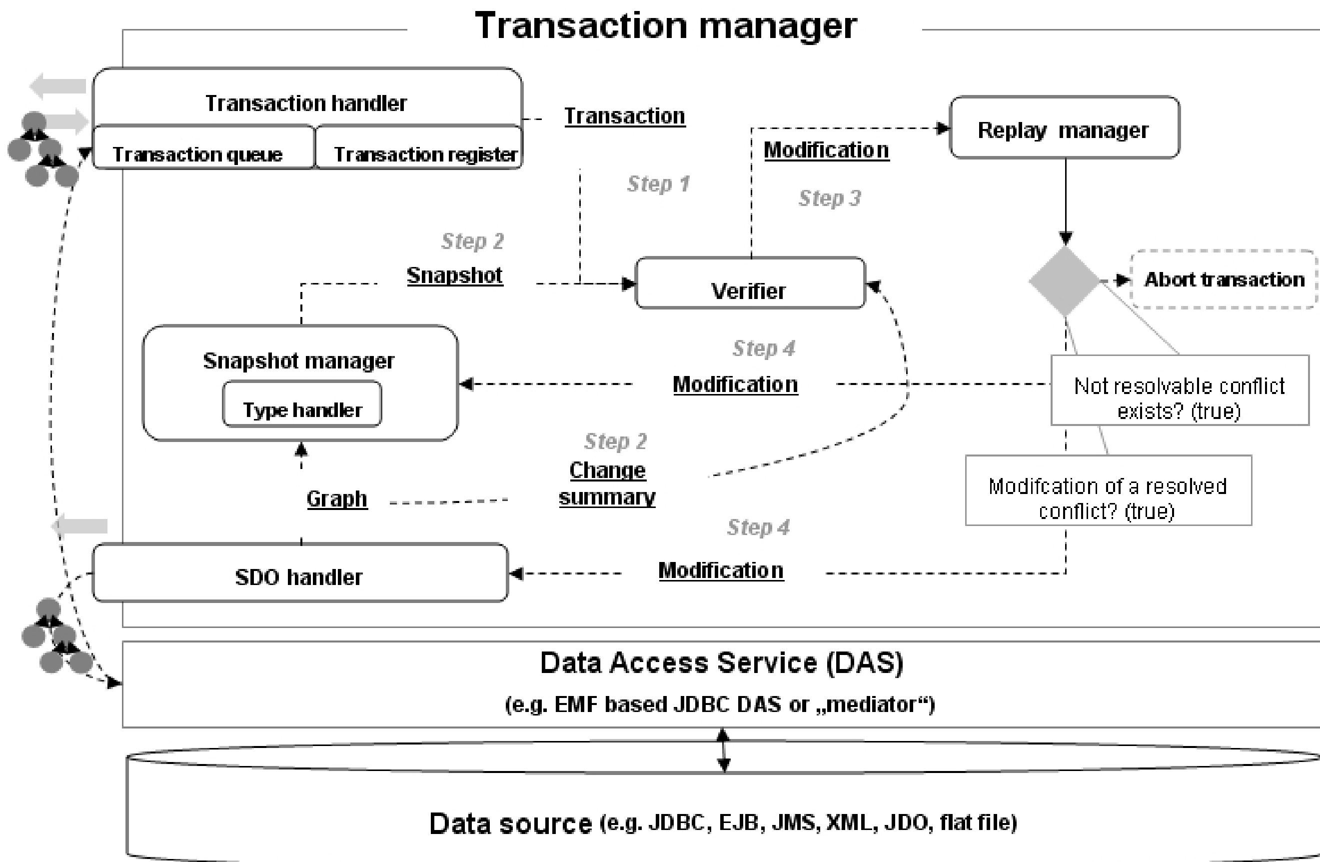

Figure 7. The transaction manager's architecture

tations come into play (Step 1, Figure 7) in the execution order $\vec{nx} \rightarrow \vec{x}$.

Each changed attribute is OCC validated against the snapshot (Step 2, Figure 7). 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 7) who handles the snapshot's modifications and the graph's changes (Step 4, Figure 7). 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.

7. Alternative conflict detection using RVV

Another approach (not sketched out in section 6) to detect conflicts at row level is the Row Version Verification (RVV) discipline (see [20]). A version indicates a change of a tuple/row and it is incremented each time the row is modified. To detect conflicts the TM reads the current version and compares the current version with the version read by the transaction. If the two versions differ the transaction is aborted, otherwise committed. RVV's advantage is a fast conflict detection at row level (DBMS level), and even modifications of non *ec* - transactions (connected or legacy) are detectable. Concerning the *ec*-model a more fine grained

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: $T_i = (op_{ik}, i = 1, 2, \cdots, k_i)$
ensure: $T' := \{T^{NX}, \mathbf{T}^{CX}, \mathbf{T}^X\}$
for $\forall\, ec_i \in \{ec_1, ec_2, \cdots, ec_n\}$ received **do**
T-number: $TNR_i := TNC; TNC++$
for $\forall\, T_j \in \mathbf{T}^{val} : TNR_j < TNR_i$**do**
  // test 1 for NX only
  if$(RSet^{NX}(i) \cap WSet(j) \neq 0)$
    abort T'
  else
  // test 2 for CX only
  ensure: $RSet(T') \subseteq D^S$
  ensure: serial execution
  if ($\exists cx$:$(c = false)$ **then**
    abort(T')
  else
    for $\forall T_i(op_n(cx)) \in \mathbf{T}^{CX}$
      //reconcile
      do $T_i(op_n(x))$
    end for
    for $\forall T_i(op_n(x)) \in \mathbf{T}^X$
      //reconcile
      do $T_i(op_n(x))$
    end for
  end if
  end if
end for
end for

$\mathbf{T}^{val}$ each validating transaction.

Figure 6. $VAL^{\neg CS}$ with reconciliation for *ec* serializability

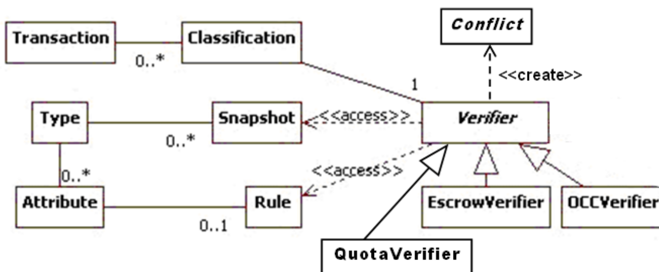

Figure 8. Abstract design of the transaction manager

detection of changes is required.

Assume the row (tupel) $(id, location, quantity, v)$, where v is the version. Now, two transactions T_1 and T_2 update $quantitiy$ concurrently, where T_1 writes first. Thus, the version is incremented and if T_2 tries to write, the TM has to abort T_2. To faciliate the TM to support a reconciliation the TM (1) has to know the row's current version v_c in order to compare v_c with the version read by the transaction. And, (2) if a conflict is detected the TM must be able to indicate a modification of $quantity$ only; Assumed $quantity$ is reconcilable. To detect fine grained modifications if a conflict was detected, requires to re-read the row with its current version and to analyze the row. Analyzing means that the TM has to correlate the changes with the matching tupel's component. If such fine grained modifications are detectable reconciliation is possible. Nevertheless, the drawback of (1) and (2) is that during the phases re-reading of the version and conflict detection, or in the case of a conflict also the phases re-read the row, analyzing it, and reconciliation, the row has to be consistent.

Now, assume the row above is divided into $(id, location, ref_q, v)$ and $(quantity, v^q)$, where ref_q is a reference on quantity. Now two versions have to be verified, but v^q directly indicates modifications for $quantity$, and prevents the analysis of the whole row.

Another concern is, if modifications, represented by an atomic change set, belong to several rows with corresponding versions. Then version verifiying has to be atomic and each version has to be compared, or another mechanism is provided (e.g. *intent versioning*). Furthermore, to enable reconciliation for each modified row the TM has also to re-read each conflicting row.

In general, the main difference is RVV tries to write on the database in order to detect conflicts. Whereas the snapshot handler detects conflicts and ensures *ec* serializability before any data is written (sent) into the DB. The snapshot handler's drawback is to ensure a synchronous and consistent state between middleware and DB level. Its advantage is a common representation for the change set, and the snapshots which alleviates conflict detection. As mentioned, RVV's advantage is a fast conflict detection at row level (DBMS level), and even modifications of non *ec* - transactions are detectable. The overhead for a fine grained error detection and the required consistence of a row have to be still analyzed.

Another solution like an event driven one, which triggers an event to notify each participating node of a modification is also conceivable. Each solution has to break through the obstacle to keep versions or snapshots synchronous across layers and during concurrent access.

In summary, RVV is an easy to understand solution and it's generally applicable on the *ec*-model, especially for nx entities. RVV itself is a more architectural aspect which may be needed in some scenarios. Versioning in principle is a well known discipline.

8. Performance of the *ec*-model

We ran a series of simulations of concurrent withdraw transactions accessing the same account. The transaction configuration parameters were as following: Reading the

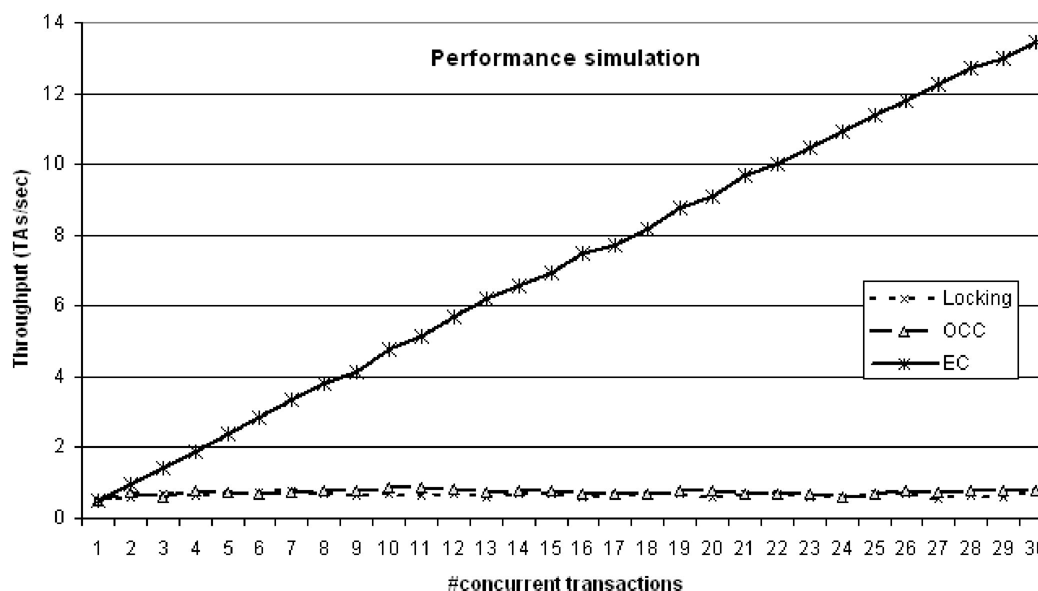

Figure 9. Performance of EC compared with OCC and locking

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 9.

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 [37] 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 4. 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 4) 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.

9. 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 pre-claiming 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.

10. Future work

The following outlines some important points of future research. Even if first simulations show higher transaction throughput, they need to become re-engineered. More complex business scenarios with several devices involved and an implementation based at databases driver level are some aspects. Such an implementation will simplify benchmarks and ease the existing SDO-implementation. Concerning the development of software, developers should be supported by methods to define or classify dependency functions for transactions which intend to use escrow serializability. In general, to define and detect a transaction's semantics manually is a drawback of many other transaction models using semantic properties. Regarding our model, an investigation of the opportunities to derive dependency functions based on common interaction patterns or on a data type base is intended. A not mentioned, but relevant concern is recovery in the escrow model, and more research has to be done in this field.

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.

References

[1] Fritz Laux and Tim Lessner. Transaction processing in mobile computing using semantic properties. *The First International Conference on Advances in Databases, Knowledge, and Data Applications, DBKDA, Cancun, Mexico*, 169, March 2009.

[2] J.H. Abawajy and M. Mat deris. Supporting disconnected operations in mobile computing. *Computer Systems and Applications, ACS/IEEE International Conference on*, 0:911–918, 2006.

[3] Shirish Hemant Phatak and Badri Nath. Transaction-centric reconciliation in disconnected client-server databases. *Mob. Netw. Appl.*, 9(5):459–471, 2004.

[4] Joanne Holliday, Divyakant Agrawal, and Amr El Abbadi. Disconnection modes for mobile databases. *Wirel. Netw.*, 8(4):391–402, 2002.

[5] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst.*, 8(2):186–213, 1983.

[6] Abdel Aziz Farrag and M. Tamer Özsu. Using semantic knowledge of transactions to increase concurrency. *ACM Trans. Database Syst.*, 14(4):503–525, 1989.

[7] Shi-Ming Huang and Chien-Ming Huang. A semantic-based transaction model for active heterogeneous database systems. *Systems, Man, and Cybernetics*, 3:2854–2859, 1998.

[8] Dimitrios Georgakopoulos, Marek Rusinkiewicz, and Amit P. Sheth. On serializability of multidatabase transactions through forced local conflicts. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 314–323, Washington, DC, USA, 1991. IEEE Computer Society.

[9] Sharad Mehrotra, Rajeev Rastogi, Henry F. Korth, and Abraham Silberschatz. Non-serializable executions in heterogeneous distributed database systems. In *PDIS '91: Proceedings of the first international conference on Parallel and distributed information systems*, pages 245–252, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[10] Shi Ming Huang, Irene Kwan, and Chih He Li. A study on the management of semantic transaction for efficient data retrieval. *SIGMOD Rec.*, 31(3):28–33, 2002.

[11] Peter Graham and Ken Barker. Effective optimistic concurrency control in multiversion object bases. In Elisa Bertino and Susan Darling Urban, editors, *ISOOMS '94: Proceedings of the International Symposium on Object-Oriented Methodologies and Systems*, volume 858 of *Lecture Notes in Computer Science*, pages 313–328, London, UK, 1994. Springer-Verlag.

[12] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[13] Fritz Laux, Tim Lessner, and Martti Laiho. Semantic transaction processing in mobile computing. In Cherif Branki, Brian Cross, Gregorio Daz, Peter Langendrfer, Fritz Laux, Guadalupe Ortiz, Martin Randles, A. Taleb-Bendiab, Frank Teuteberg, Rainer Unland, and Gerhard Wanner, editors, *TAMoCo*, volume 169 of *Frontiers in Artificial Intelligence and Applications*, pages 153–164. IOS Press, 2008.

[14] Qi Lu and M. Satyanaranyanan. Isolation-only transactions for mobile computing. *Operating Systems Review*, 28:81–87, 1994.

[15] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[16] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA, 1996. ACM.

[17] G. D. Walborn and P. K. Chrysanthis. Supporting semantics-based transaction processing in mobile database applications. In *SRDS '95: Proceedings of the 14TH Symposium on Reliable Distributed Systems*, page 31, Washington, DC, USA, 1995. IEEE Computer Society.

[18] Gary D. Walborn and Panos K. Chrysanthis. Transaction processing in pro-motion. In *SAC '99: Proceedings of the 1999 ACM symposium on Applied computing*, pages 389–398, New York, NY, USA, 1999. ACM.

[19] Rainer Unland. Optimistic concurrency control revisited. Technical report, Department of Business Informatics, University of Muenster, 1994.

[20] Martti Laiho and Fritz Laux. Data access using rvv discipline and persistence middleware. *e RA - 3, Greece*, 2008.

[21] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[22] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 5th Edition*. McGraw-Hill Book Company, 2006.

[23] K. A. Momin and K. Vidyasankar. Flexible integration of optimistic and pessimistic concurrency control in mobile environments. In Julius Stuller, Jaroslav Pokorný, Bernhard Thalheim, and Yoshifumi Masunaga, editors, *ADBIS-DASFAA '00: Proceedings of the East-European Conference on Advances in Databases and Information Systems Held Jointly with International Conference on Database Systems for Advanced Applications*, volume 1884 of *Lecture Notes in Computer Science*, pages 346–353, London, UK, 2000. Springer-Verlag.

[24] Stefano Ceri and Susan S. Owicki. On the use of optimistic methods for concurrency control in distributed databases. In *Berkeley Workshop*, pages 117–129, 1982.

[25] Ho-Jin Choi and Byeong-Soo Jeong. A timestamp-based optimistic concurrency control for handling mobile transactions. In Marina L. Gavrilova et al., editor, *ICCSA (2)*, volume 3981 of *Lecture Notes in Computer Science*, pages 796–805. Springer, 2006.

[26] Adeniyi A. Akintola, G. Adesola Aderounmu, A. U. Osakwe, and 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.

[27] V. Krishnaswamy, D. Agrawal, J. L. Bruno, and A. El Abbadi. Relative serializability: An approach for relaxing the atomicity of transactions. *SIGMOD/PODS 94*, 1994.

[28] Klaus R. Dittrich and Stella et al. Gatziu, editors. *Aktive Datenbanksysteme*. dpunkt Verlag, Heidelberg, BW, GER, 2000.

[29] J. E.B. Moss. *Nested transactions: an approach to reliable distributed computing*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1985.

[30] Kyong-I Ku and Yoo-Sung Kim. Moflex transaction model for mobile heterogeneous multidatabase systems. In *RIDE '00: Proceedings of the 10th International Workshop on Research Issues in Data Engineering*, page 39, Washington, DC, USA, 2000. IEEE Computer Society.

[31] Sushil Jajodia and Larry Kerschberg, editors. *Advanced Transaction Models and Architectures*. Kluwer, 1997.

[32] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[33] Katharina Hahn and Heinz Schweppe. Exploring transactional service properties for mobile service composition. In Markus Bick, Martin Breunig, and Hagen Höpfner, editors, *MMS*, volume 146 of *LNI*, pages 39–52. GI, 2009.

[34] M. Adams and C. Andrei et al. Service data objects for java specification. *OSOA (BEA Systems, IBM, et al.)*, 2.1, 2006.

[35] J. Beatty, S. Brodsky, M. Nally, and R. Patel. Next-generation data programming. *BEA Systems, IBM*, 2003.

[36] Tim Lessner. Transaktionsverarbeitung in disconnected Architekturen am Beispiel von Service Data Objects (SDO) und prototypische Implementierung eines Transaktionsframeworks. Diploma thesis (ger), 2007.

[37] TPC. Tpc benchmark c, standard specification, revision 5.9, 2007.