

# Efficient Consistency Support for Distributed Mobile Applications \*

© Anna Kozlova

© Dmitry Kochnev

© Boris Novikov

University of Saint-Petersburg  
anet@ntc-it.ru

University of Saint-Petersburg  
dmitry@smartphonelabs.com

University of Saint-Petersburg  
borisnov@acm.org

## Abstract

This paper presents a model of the distributed middleware with transactional support. Our approach provides for high availability of the system in the fluctuated mobile environment and a high degree of a consistency when network connections are stable. The proposed set of the high-level operations allows high level of concurrency while processing XML-like data structures. A concept of the "accumulator" is introduced, providing for efficient conflict resolution during reconciliation.

## 1 Introduction

Business goes mobile. Cellular mobile networks allow for high-speed data transfer. Cellular networks coverage becomes more and more comprehensive. The number of high performance mobile devices which support complex applications is increasing from year to year. One of the most promising directions of the mobile IT-industry [11] is the development of applications with the vertical architecture known as business-to-employee (B2E) applications. Such applications provide solutions for interaction problems between the company and its employees. Within these applications, it is possible to classify the following systems:

- Enterprise Resource Planning, ERP;
- Customer-Relationship Management, CRM;
- Customer Information System, CIS;
- Sales Force Automation, SFA;
- Supply-Chain Management, SCM;
- Team-Management System, TMS

The basic problem which should be solved when creating of a B2E-system is to adapt a well-known design of wired solution which uses wired business-processes for the mobile environment. There are two approaches for the problem of adaptation highlighted in this article: application-transparent and application-aware adaptation [21]. The other radical approach frequently uses in the industrial applications: the conflicts that arises after disconnection in such applications, are being resolved using one of the

predetermined rules, for example, the user should choose a solution for each conflict manually.

### 1.1 Constraints of Mobility

The mobile environment imposes a number of restrictions on the applications. One group of restrictions is related to communication characteristics of a cellular network, the other group is induced by the specificity of mobile devices.

Performance and communication characteristics of a cellular network may fluctuate depending on a place, time of day, weather, activity of subscribers and many other factors. A stable cellular network, as we define it possesses the following characteristics:

- *good communication characteristics, i.e. "big enough" data amounts may be transferred "quickly enough" between the client and the server*
- *intermittent network connection, i.e. network connection can not be kept alive for a long time.*

Mobile devices which supports complementary applications (e.g. java or BREW), represent the compact devices connected to a cellular network, such as Java-phones, smartphones, and PDA-like devices. All these devices have the following properties:

- *limited sizes of memory and storehouse of the data*
- *small computer power and small power resources*
- *allows work in the disconnected mode (disconnected operation availability),*
- *may connect to the server via some protocol*
- *the number of such clients can be huge*

### 1.2 Problem statement

The system which is considered in the scope of this paper has nomadic architecture [15]. It means that clients communicate through mobile network with the primary server. We assume that clients do not communicate directly among themselves. The primary server is a high performance computer that has enough of memory and computational resources to manage plenty of connections and large amounts of data.

This paper is focused only on the problems concerning middleware implementation for the distributed nomadic system based on some classical database. In nomadic systems mobile devices move from location to location, while maintaining a

connection to the fixed network. Usually, the wireless network connects the edges of a fixed infrastructure to the mobile devices. [15]

The nature of a database is not important. Regarding client application, technological restrictions and features related to implementation of client application for such system are described in detail in [12].

According to specifics of a cellular network, it is necessary to store a replica of some data from the primary server on the client, to increase the availability of the data. It allows client disconnected operation between connection losing and recovering to a primary server. The purpose of this paper is to design the prototype of the middleware for the mobile distributed system with the support of transactions which provides the following:

- high level of availability of system in the unstable (fluctuated) mobile environment;
- high degree of a consistency of the data in the environment of a stable mobile network.

## 2 Related Works

The distributed systems may be distinguished as *traditional* distributed systems, *nomadic* distributed systems, and *ad-hoc* mobile distributed systems. The review is focused on the concept of the middleware for nomadic systems. The detailed middleware review is resulted in [15]

### 2.1 Classical Middleware

Most of existing middleware technologies, such as object-oriented middleware [6], and also message- and transaction-oriented systems, hide from the application the heterogeneity and the distributed nature of its environment. This approach being very efficient and cost-effective in "wired" environment does not suit for the systems that work in wireless environment [2]. The basic problems arising at moving of the system to a mobile environment are synchronization of the data (i.e. restoration of data consistency) and maintenance of availability (inapplicability of data locking) [10].

### 2.2 Data Sharing Middleware

One of the major issues targeted by data-sharing middleware systems such as Bayou [5], Coda [23], its successor Odyssey [22] and Xmiddle [16] is the support for disconnected operation and data-sharing.

The base architectural principles of the proposed solution such as flexible client-server architecture supporting low-resource mobile clients, conflict resolving on the application layer, maintenance of a master replica, processing disconnected operation on the tentative replica, etc. are inherited from the Bayou architecture concept [5]. Instead of Bayou, the proposed solution supports the correct histories on the clients and on the server and allows processing the reconciliation and further replaying of histories instead of applying replica merging.

Coda distributed file system [23] provides resilience to server and network failures through two mechanisms:

server replication and disconnected operation. Odyssey [22] improves the application transparent approach by Coda introducing context-awareness and application-dependent behaviors. But, Odyssey suffers from some limitations: the data that can be moved across mobile devices may be too coarse-grained. The lack of semantics in the files complicates the development of conflict detection and reconciliation policies from an application point of view. As against the above, offered system focuses on efficiency of data granularity and improves Odyssey's architectural solution on application-aware approach by offering server-based accumulator abstraction.

Xmiddle [16] allows mobile hosts to share data when they are connected, or replicate the tree-like data and perform operations on them off-line when they are disconnected. Reconciliation policies are specified as part of the XML Schema definition of the data structures that are handled by Xmiddle itself. The weak of that system is that the policies of reconciliation are statically defined and strongly related to the data structures are using by the application. We improve this approach so that the application may contain additional server-side units implementing conflict resolution policies as accumulators.

### 2.3 Tuple-space Systems

Tuple space based systems for logical and physical mobility such as JavaSpaces [7], Lime [17], Limbo [4], and T Spaces [9] exploit the decoupling in time and space of these data structures in the mobility context. Tuple-spaces are multi-sets, which means that every tuple can be duplicated in the space. Tuple spaces are very general concept that loses data structures, so this approach cannot be applied to the highlighted problem because it has irresolvable disadvantages with data reconciliation.

### 2.4 Mobile Transactions

For a mobile environment the classical concept of transaction is not applicable [2], [10]. There are some approaches taking into account specificities of the mobility, expanding classical definition. Most approaches use replication the data on a client part of the application which raises data availability. Pessimistic approaches [14] do not support replication and disconnected operation, so cannot be applied to the highlighted problem.

The model described in this paper uses the optimistic approach. The close approaches are represented in [8] and [19] where the commitment processed on a local replica, and then the lifecycle of the application proceeds in the assumption that the commitment will be confirmed by the server. In [8] the results of the same transaction processing on the client and server sides may differ; and the correctness of the result defines by client application.

As well as the approaches using broadcasting [20], [25], our approach reduces ascending traffic (from the client to the server). For this purpose read-only

transactions are being committed on the client side without any confirmation from the server. The local serialization graph is used for recognition of transactions that in any way cannot be serialized on the server. The similar ideas are represented in the approach of [3] where the server computes so called dependency information and broadcasts it to clients.

The detailed review focused on the computational model and ACID-properties of approaches to mobile transactions are represented in [24].

### 2.5 Data Model

Instead of Document Object Model (DOM) [13] the offered data representation uses unique identifiers for each node due to distributed and replicated nature of the data.

There are a lot of data models known for a long time that use the abstract high-level operations defined as sequences of atomic operations. The concept of multi-level transaction [26], [27] is most interesting for our purpose. In most cases the specially defined high-level operations commute with each other in spite of the elementary operations laying in their basis. Use of the similar approaches raises concurrency of transactional operation in the system.

The data considered in this paper may have a non-numerical nature and complex internal structure, therefore some semantic conflicts arising between operations can not be represented as pseudo-conflicts, and thus universal commutativity can not be achieved as in pseudo-conflicts between Withdraw() and Deposit() operations working with account [26]. The decided problem is to define the operations so that the number of irresolvable conflicts would be as little as possible.

In [18] the ideas of the concurrent video model related to high-level operations and unique identifiers applied to the linear data model. We admit that the further development of these ideas will allow applying the similar approach to the XML-structures.

## 3 Architectural Solution

### 3.1 Extended Client-server Model

The conceptual scheme of the system is represented on Figure 1: mobile clients communicate with the applications server via HTTP through a wireless channel (e.g. GPRS- or UMTS-connection) and the application server has a permanent link to the database server. Clients are typical mobile devices with limited resources intermittently connected to the Internet.

The server part of the system consists of several layers: the database, the middleware and a server part of the mobile application by third party developer. The client part is multi-layer too. It consists of the middleware that manages local replica and cooperates with a server part of the middleware through a wireless

connection: and a client part of the mobile application by third party developer.

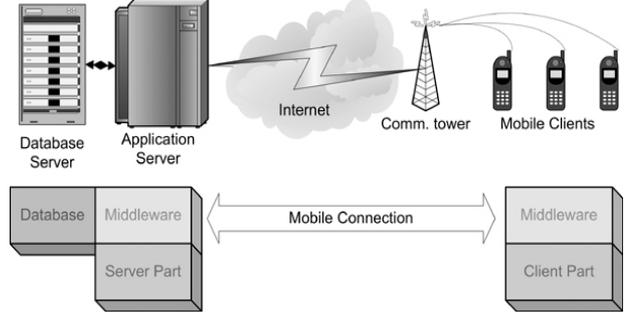


Figure 1. Conceptual Scheme

### 3.2 System Lifecycle

The typical scenario of interaction between hosts (Figure 2) gives a large scale view of the system work. First, the client subscribes for the data and receives from the server a replica marked with a special *client replica timestamp*, because clients' clocks are asynchronous with the server's clocks.

*Definition: Client's replica timestamp*

The client's replica timestamp denotes a timestamp of last connection to the server with participation of the given client.

When disconnection occurs, the client continue working with its replica. Further, at some moment depending on the adaptation policy of the application, the client restores the connection and send its changes to the server. The server reconciles the information received from the client with the master replica and returns all changes that modify client's replica. Then client reconcile the received changes to its local replica. The details on the server and client activities are described further in the appropriate sections.

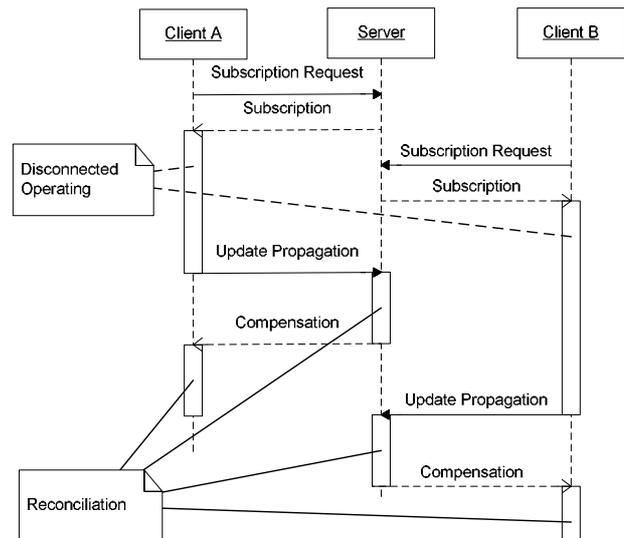


Figure 2. System Lifecycle

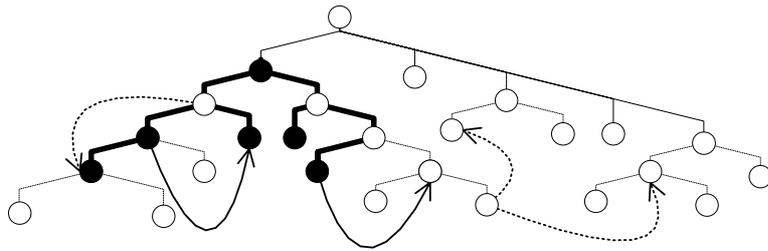


Figure 5.1. Clients replica as a part of server's replica

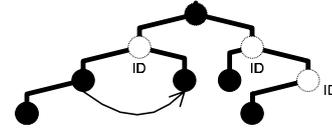


Figure 5.2. Replica on the client side

### 3.3 Nature of the Data & Related Structures

The data using by most B2E -applications (e.g. sales force automation) may be represented as tree-like structures semantically associated to graphs are stored in XML documents [28]. Naturally, the organization consists of departments; employees work in departments; each department develop some projects, related to a client base; each projects relates to a group of employees; clients are related to the projects, etc. Tree structure allows sophisticated manipulations due to the different node levels, hierarchy among nodes, and the relationships among the different elements which could be defined.

The data used in the internal representation turn out from the normal XML by applying the transformation that moves the attributes into the affiliate nodes (Figure 3). Thus, the target XML has the following properties:

- Non-leaf nodes contain the navigational information only.
- The data values are stored only in the leaf-nodes.
- The references are stored only in the leaf-nodes. Each reference-node points not more than one node. Also, the reference-nodes point to navigational nodes only.

| Navigation-Node       | Data-Node  | Ref-Node      |
|-----------------------|------------|---------------|
| -ID                   | -ID        | -ID           |
| -CHILDREN_ID_SET []   | -DATA      | -PARENT_ID    |
| -PARENT_ID            | -PARENT_ID | -REFERS_TO_ID |
| -REFERED_BY_ID_SET [] |            |               |

Figure 3. Node structure

Moreover, each node is assigned to the unique identifier due to distributed and replicated data representation. This unique identifier is kept in all replicas.

A replica presents determined by a subscription a subtree of the tree of the master replica (Figure 5.1, 5.2). If the subscription contains a pair of nodes, so that one node is an ancestor of another, then nodes of intermediate generations also will be included in a subscription, however on the client stores only identifiers of such intermediate nodes.

By default, the data in leaves is storing according to the concept of application-transparent adaptation [21]. From the application's point of view the data elements has a simple type (e.g. a number or a string). Besides this, the system allows to organize access to some data

items according to the application-aware concept. For this purpose the leaf of a tree should detail the "accumulator" abstraction offered by us in Section 5.

## 4 Operations

In the model described in this paper the following operation above tree structures are defined: *insert\_node*, *insert\_data*, and *insert\_ref*, *delete*, *update*, *select* and *move*. The operations defined such way are commutating with each other in most cases. The formal definitions are presented further.

*Definition: Insert operation*  $Insert(parent\_id, sibling\_id): newID$

This operation inserts a new node in the data tree as a child of the node with *parent\_id* identifier and as a next sibling of the node with *sibling\_id* identifier. If *sibling\_id* is not set, the operation creates the first child of the node with *parent\_id* identifier. If any required node is not found or has inappropriate type, this operation reports failure.

*Definition: Insert operation for navigation nodes*  $InsertNode(parent\_id, sibling\_id): newID$

This operation inserts a navigation node using *Insert()* operation.

*Definition: Insert operation for data nodes*  $InsertData(parent\_id, sibling\_id, data): newID$

This operation inserts a data node using *Insert()* operation according to the following algorithm:

1. Execute *Insert (parent\_id, sibling\_id)*
2. Associate the created node to the *data* entry

*Definition: Insert operation for reference nodes*  $InsertRef(parent\_id, sibling\_id, toID): newID$

This operation inserts a reference node using *Insert()* operation according to the following algorithm:

1. Execute *Insert (parent\_id, sibling\_id)*
2. Find a node with *toID* identifier. Report failure if search failed
3. Associate the created node to the node with identifier *toID*
4. Insert the identifier of created node into *REFERED\_BY\_ID\_SET*

**Table 1: Commutativity of proposed operations**

|            | InsertNode | InsertData | InsertRef | Update | Delete | Select | Move   |
|------------|------------|------------|-----------|--------|--------|--------|--------|
| InsertNode | TRUE       | TRUE       | TRUE      | ALWAYS | TRUE*  | TRUE   | TRUE   |
| InsertData | TRUE       | TRUE       | TRUE      | TRUE   | TRUE*  | TRUE   | TRUE   |
| InsertRef  | TRUE       | TRUE       | TRUE      | ALWAYS | TRUE*  | (1)    | TRUE   |
| Update     | ALWAYS     | TRUE       | ALWAYS    | TRUE   | TRUE*  | TRUE   | ALWAYS |
| Delete     | TRUE*      | TRUE*      | TRUE*     | TRUE*  | ALWAYS | (2)    | TRUE*  |
| Select     | TRUE       | TRUE       | (1)       | TRUE   | (2)    | ALWAYS | TRUE*  |
| Move       | TRUE       | TRUE       | TRUE      | ALWAYS | TRUE*  | TRUE*  | ALWAYS |

**The Legend:**

- Additional assumptions marked as star follows:  
 (\*)  $Op_1(X) \circ Op_2(args) == Op_2(args) \circ Op_1(X)$  only if  $X \notin Parent(args)$  where  $Parent(args)$  is a conjunction of parent sets of all elements of  $args$ , and  $Op_1$  is  $Select()$  or  $Delete()$ .
- Additional conditions dependent on the parameter values of the operations (and independent on database state) are required in the cases marked as (1) and (2).

*Definition: Update operation for data-nodes  $Update(id, new\_data)$*

This operation sets the data entry associated to the node with  $id$  identifier to  $new\_data$ . If the required node is not found it reports failure.

*Definition: Delete operation  $Delete(id)$*

This operation removes the full subtree with the root node identified as  $id$ .

1. Find a node with identifier  $id$ . Set it as *current* node
2. Remove *current* node from the  $CHILDREN\_ID\_SET$  of its parent node
3. If this node is a reference ( $REFERS\_TO\_ID$  is not empty) then removes *current* node from  $REFERED\_BY\_ID\_SET$  of the node with  $REFERS\_TO\_ID$  identifier
4. Apply  $Delete()$  recursively for each reference nodes listed in  $REFERED\_BY\_ID\_SET$  of the current node
5. Apply  $Delete()$  recursively for each node listed in  $CHILDREN\_ID\_SET$  of the current node
6. Remove the current node

*Definition: Select operation  $Select(id)$*

This operation returns the node with identifier  $id$  as an object or reports failure if the required node does not exist.

*Definition: Move operation  $Move(id, new\_parent\_id, new\_sibling\_id)$*

This operation set the subtree with root node identified by  $id$  as a child of the node with  $new\_parent\_id$  identifier and as a next sibling of the node with  $new\_sibling\_id$  identifier. If  $new\_sibling\_id$  is not set, the operation inserts the first child of the node with  $new\_parent\_id$  identifier. If any required node is not found or has inappropriate type, this operation reports failure. The provisional algorithm follows:

1. Find nodes with identifiers  $id, new\_parent\_id, new\_sibling\_id$ . Report failure if such nodes do not exist.

2. Assign a node with  $new\_parent\_id$  identifier as a parent to a node with identifier  $id$
3. Insert  $id$  value after  $new\_sibling\_id$  into the  $CHILDREN\_ID\_SET$  of the node with  $new\_parent\_id$  identifier.
4. Remove identifier  $id$  from  $CHILDREN\_ID\_SET$  of the old parent of  $id$

**4.1 Commutativity Table**

The following table describes the commutativity aspects of proposed operations. The pairs of operations that always commute marked in the table as *ALWAYS*.

If the disjunction of the argument sets of both operations is empty, the pairs of operations marked in the table as *TRUE* will commute. Naturally, the commutativity between some operation and  $insert*()$  requires the weaker assumption: the *NewID* generated by  $insert*()$  should not equal to any argument of the second operation.

Thus, as it was shown, the proposed high-level operations in most cases commute with each other in spite of the elementary read and write operations laying in their basis.

*Example*

The operations  $Move()$  and  $InsertNode()$  may be represented as a composition of elementary read-write operations as follows:

$$\begin{aligned}
 Move(id, new\_parent\_id, new\_sibling\_id) &\cong \\
 &\cong Read(id) \circ \\
 &\circ Read(new\_parent\_id) \circ \\
 &\circ Read(new\_sibling\_id) \circ \\
 &\circ Write(id \rightarrow getParentID()) \circ \\
 &\circ Write(new\_parent\_id) \circ \\
 &\circ Write(id);
 \end{aligned}$$

$$\begin{aligned}
 InsertNode(parent\_id, sibling\_id): newID &\cong \\
 &\cong Read(parent\_id) \circ \\
 &\circ Read(sibling\_id) \circ \\
 &\circ Write(parent\_id) \circ \\
 &\circ Write(newID);
 \end{aligned}$$

If  $id == parent\_id$  there is a conflict between elementary operations, instead of it, the proposed high-level operations commute with each other.

## 5 Accumulators

Accumulators are intended for "intervention" of the application to the conflict resolution. As against simple types, the accumulator stores operations applied to some initial value of simple type instead of the explicit value of the element. Thus, accumulators allow on the air replaying the operations. It decreases number of conflicts between *update* operations, i.e. reduces number of transaction aborts in the system.

*Definition: Base Value of Accumulator*

The *base value* is value of simple type. To compute the value of the accumulator on any moment of time one shall apply the operations stored in accumulator to that base value.

*Definition: Operations Collection of Accumulator*

The accumulator stores operations in a collection. Each operation in the accumulator has two timestamps. The insertion timestamp defines the moment when this operation has been added into the accumulator; and the other one is execution timestamp that equals to the client replica timestamp (because from server's point of view the client has no clock). The operations are defined and implemented at the application side.

*Definition: External Function of Accumulator*

*External functions* are defined at the layer of application by third party developer. External functions are intended to get additional information about the accumulator depended on some moment or on some period of time. External functions may depend not only on operations and base value of the accumulator, but also on other accumulators. If an external function semantically depends on the period of time, it means that this function depends on operations that have been applied during this period.

*Example*

For example, "daily average balance on the account" may be an external function for the account represented as accumulator. The value of this function can be used for operation Op defined as "monthly interest of the account". The accumulator allows inserting into the history transactions that could not be inserted according to application-transparent approach. If after applying of the operation Op the server receives the operation Op1 that influence to the result of applying Op, such operation will be just added into the accumulator. As a result, the current value provided by the accumulator, will allow for Op1 applying, because this value is calculated "on the air".

The accumulator is located on the server side only, and only the calculated value of the accumulator transferring to a client part of the application.

### 5.1 Accumulator Design Details

Two typical designs of the collection are introduced; they are list- and set-accumulators. The application developer may also implement his own designs of the collection. The main difference between list- and set-accumulator collections is that in a list all elements are strictly ordered and in the set all elements are stored without any order.

Uniqueness of elements in set is not important in this context, because each operation received by the server has unique identifier. This identifier consists of unique identifier of the client that causes this operation and unique identifier of this operation on that client.

For each type of collection one of the proposed strategies of modification may be applied: the first strategy is so called insert-only strategy; the second one is insert-and-remove strategy. The methods of replaying the operations for both strategies are defined further. The application developer may choose between those methods according to time-dependence of external functions and conflict presence between defined operations.

#### 5.2 Set-Accumulator

The time concept cannot be applied to sets, because it would define the artificial order. Also, the values of the external functions do not depend on the order of elements; it means that each operation in the set should commute with each other.

Inserting or removing the new operations in such accumulator occurs as inserting or removing the element in the set. Replaying of some operation in set can be implemented by two methods: as replacement of old operation with the new one, or as inserting a compensational operation and then inserting of the new operation.

Because of commutativity of operations, the results of each method are the same. The only difference is that in the first case the removed operation will not appear anymore in the accumulator; in the second case inserting the compensational operation requires its explicit existence.

#### 5.3 List-Accumulator

Because of strictly defined order, each operation stored in the list-accumulator has a context of other operations. (By the way, the list-accumulator may be applied as a set, because each operation has its unique identifier, and the context may be ignored in the application.) There are two kinds of list-accumulators: insert-only and insert-and-remove list-accumulators. Inserting or removing the new operations in such accumulator occurs as inserting or removing the element in the set.

The replaying strategies for the list-accumulator follow.

*Strategy 1 Insert-only Strategy for the List-accumulator*

An insert-only list-accumulator supports only one way for replaying: insertion of a compensational operation and insertion of the new operation. There are following

cases related to time-dependency of the external functions and the commutativity of the defined operations:

*Case 1: There are some time-dependent external functions in the accumulator*

i. All operations in the accumulator commute with each other

Inserting of the compensational and new operations occurs according to the rules of detection of the execution timestamps.

ii. Some operations in accumulator do not commute with each other

Inserting of the compensational and new operations occurs according to the rules of detection of the execution timestamps. Because the operations do not commute, the application should warrant the semantic correctness of the replaying.

*Case 2: All external functions are time-independent*

i. Some operations in accumulator do not commute with each other

Inserting of the new element in any place of the list does not damage the rules of detection of the execution timestamps because all external functions are time independent. Thus, there are several cases to replay the operations:

- Insertion of the compensational operation and the new operation just after the old operation
- Insertion of the compensational operation just after the old operation and inserting the new operation according to its execution timestamp
- Insertion of the compensational and the new operations according to their execution timestamps

The compensational operation and the method of replaying are defined by the application.

*Strategy 2: Insert-and-Remove Strategy for the List-accumulator*

An insert-and-remove list-accumulator support two approaches for replaying: inserting of the compensational operation (the same way as insert-only case, considered above) and removing of the old operation. If some external function depends on time some information related to the old operation may be lost, so this method damages the semantic correctness of such function.

*Case 1: All external functions are time-independent*

i. Some operations in accumulator do not commute with each other

Inserting of the new element in any place of the list does not damage the rules of detection of the execution timestamps because all external functions are time independent. Thus, there are several cases to replay the operations:

- Removal of the old operation and inserting of the new one with the same execution timestamp
- Removal of the old operation and inserting of the new one with the new (actual) execution timestamp

The application defines the replaying method.

#### 5.4 Purging of Out-of-Date Operations

While the system works the number of the operations in the accumulators grows. To improve the performance of the system, it is necessary to purge out-of-date operations and to replace the base value with the new one from time to time. The new value is a result of applying the operations that will be purged to the old value, but generally

$$F(b, (op_1, op_2, op_3)) \neq F(op_1(b), (op_2, op_3)) (*)$$

This issue may be solved using additional assumption about the external function: it must depend on limited (explicit) period of time, i.e. depend on final number of operations. This period of time refers to the dependent window of the external function. Of course, this number may be great. Thus, the operation became out-of-date if its execution timestamp  $t$  satisfies to the following predicate:

$$t < t_{current} - \max_{i=1..m} \{t_i \mid t_i - \text{dependent window of } F_i\}$$

#### 5.5 Accumulators Advantages

The offered strategies allow the application to define the criteria of a correctness using definition of the operations and external functions. Thus, the application is permitted to influence resolutions of conflicts.

Purging of the accumulator from old operations also may be initiated by the application, it allows the server to manage the instance of accumulator in the efficiently manner.

### 6 Protocol for mobile transactions

The proposed protocol is based on low-level protocol that supply data transfer between mobile hosts, for example, HTTP over GPRS or UMTS. The protocol must satisfy to the following conceptual requirements:

1. The server must reconcile histories from all clients and keep the master replica in consistent state.
2. The clients must propagate their changes to the server and receive from the server the changes made by the other clients that modify the local replica of this client.
3. To achieve the appropriate level of availability the client should operate in disconnected mode.
4. During disconnected operation the client should manage two local replicas of the data. The *primary replica* one should be identical to the master replica stored on the server at the moment of the last connection to the server. All changes during disconnected operation should be applied to the *tentative replica*.

5. The *local conflicts* (the conflicts on the tentative replica) must be resolved by the client without any participation from the server.
6. The client may commit read-only transactions. Commitment of read-only transactions at the client side has several advantages, such as reducing transaction commitment delays at the client side, reducing communication costs, and limiting the processing load of the server [3].
7. To save its computational resources, the client should process the garbage collection, i.e. to clean the memory from the unused structures for conflicts resolving on tentative copy.

The large scale of the protocol is shown on the Figure 6.

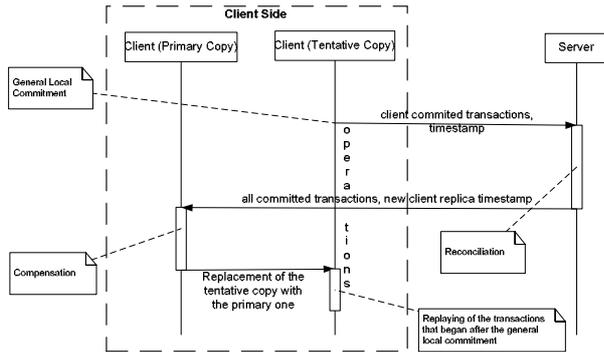


Figure 6. The Large Scale of the Protocol

### 6.1 Client Side Algorithm

1. Disconnected operation
  - Permanent concurrent execution of the transactions on the tentative copy and manage the local serialization graph
  - If the server response that was requested in the previous execution of the step #2 has been received, go to step #3
  - After returning from the step #3 and #4 may continue with the step #2
2. Updates propagation
  - Pause disconnected operations (Step #1)
  - General local commitment processing, i.e. commitment of all local transactions
  - Purge all update transaction from the serialization graph
  - Mark all read-only transactions
  - Send the *commit\_request* that includes all update transactions and the client replica timestamp to the server
  - Continue with the Step #1
3. Compensation
  - Receive the part global history that modifies the client's replica
  - Note, that the local transactions will be added to the end of this history
  - Add the received transactions to the local serialization graph

- Cascadely abort of all transactions that reads from transactions that have been aborted by the server
- Cascadely abort of all transactions that conflicts with transactions that have been committed globally

### 4. Reconciliation

- Replay the history received from the server on the primary copy
- Replace the tentative copy with the primary copy
- Pause disconnected operations (Step #1)
- Purge the serialization graph from all transactions with server timestamps and from read only transactions marked on the Step #2
- Replay the transactions from the serialization graph on the tentative copy
- Continue the disconnected operations (Step #1)

### 6.2 Server Side Algorithm

1. Receive the transactions from the client in serial order with the client replica timestamp
2. Insert the received transactions into the global history using histories merging procedure
3. Send to the client all transactions that modify the local replica of that client and have not been sent to him yet. Send the list of local transactions that have been aborted by the server

#### Histories merging procedure

The following algorithm is intended to the operation with non-accumulator arguments. Otherwise the accumulators' technique should be applied.

The operation *op* having timestamp *t* can be inserted into the history *H* if

$$\forall op(i, x_i, t_i) \in H, t_i > t(op(i, x_i, t_i) \circ op \equiv op \circ op(i, x_i, t_i))$$

If it is true for each operation in the transaction then this transaction can be serialized.

The policy the transaction aborting may be defined on the application side. For example, it is possible to abort a transaction with the minimal timestamp. This algorithm may be improved using the method [1], where for each data element the timestamp of last read, write etc. are stored in the special records.

#### Approach to Information Selection for Transferring to the Client

To determine what transactions should be sent to the client, the server uses the *client replica timestamp*. If the pre-image of the client's replica at the moment of the client replica timestamp on the server does not contain any elements of accumulator type, the *client replica timestamp* uniquely determines the data of this pre-image at the moment defined by the client replica timestamp. Thus the server should send to the client all operations from the global history that has a timestamp greater than the *client replica timestamp* and that has the

arguments from the pre-image of the client's replica at the moment of the *client replica timestamp*.

The previous approach cannot be applied if the pre-image of the client's replica at the moment of the *client replica timestamp* on the server contains some elements of accumulator type. For example, after the moment of the client A replica timestamp the client B inserts an operation with the *execution timestamp* less than the client A replica timestamp, and this operation does not commute with the subsequent operations. So, the value of the accumulator in the moment defined by the client A timestamp was changed, but it cannot be propagated to the client using the previous approach.

In the case of elements of accumulator type the following approach is applied for reconciliation client's replica with the server's data. The server sends to the client not only the operations but also the values of the element having accumulator type at the moment of the client replica timestamp only if there is an operation stored in the accumulator that is satisfying to the following predicates:

$$\begin{aligned} & \text{client replica timestamp} > \text{execution timestamp} \\ & \text{client replica timestamp} < \text{insertion timestamp} \end{aligned}$$

This value should be included into the client's primary copy. Use this approach warrant the appropriate level of consistency of the client and the server replicas.

To save the computational resources of the client host it is possible to transfer the whole new client's replica together with the operations. However it additionally loads the network.

### 6.3 Correctness of the protocol

*Lemma 1.*

Inserting the read-only transactions into a global history does not influence its serializability:

$$\begin{aligned} & \forall H_1 - \text{the serial history of } C_1 \\ & \forall H_2 - \text{the serial history of } C_2 \\ & \exists H - \text{the serial history: } H \supseteq H_1 \wedge H \supseteq H_2 \end{aligned}$$

*Proof:*

1. Note that  $H_1$  and  $H_2$  differ only with the read-only transactions.
2. According to the proposed protocol the history  $S_1 := (Op(H_1) \setminus \{op \in Op(H_1) \mid op \in \text{some read-only transaction of } H_1\}, <_{H_1})$  is equivalent to  $S_2 := (Op(H_2) \setminus \{op \in Op(H_2) \mid op \in \text{some read-only transaction of } H_2\}, <_{H_2})$  (as it defined in [1]) and these histories are equivalent to some serial history according to the server side algorithm.
3. Insert all read-only transactions of  $H_1$  and  $H_2$  to  $S_1$  (and denote it as  $S_1'$ ) and to  $S_2$  (and denote it as  $S_2'$ ).
4. Since read operations do not conflict with each other,  $S_1'$  and  $S_2'$  are equivalent.

5. So, the global history  $H_1'$  containing all transactions of all clients has been found. Also, this history is equivalent to some serial history.

*Lemma 2*

The purging of the serialization graph on the client side is correct, i.e. the transactions that would begin after the purging would not conflict with transactions have been removed from the graph.

*Proof:*

On the client side the new transactions that began after the general local commitment has been processed are succeeding the transaction that has been purged from the local serializing after the commitment, so that any transaction will not go through local commitment. Thus, the purged transactions do not influence the processing of new transactions during disconnected operation.

Further, on the server side new transactions will receive the timestamp of the moment of the global commitment of the transactions purged on the client side before new transactions started. Thus, from the server point of view the new transactions are also succeeding the transactions that were locally committed on the client before.

*Statement: General Local Commitment Assumption*

Prove that the assumption of general local commitment of all transactions on the client side is essential. It means that it is impossible to warrant that purging of the serialization graph would not damage its integrity.

Consider the following example: the edge  $t_1 \rightarrow t_2$  of the serialization graph was assigned to of the conflict between transactions  $t_1$  and  $t_2$  cause by operations  $q_1(x)$  and  $p_2(x)$ . The transaction  $t_2$  has been committed and purged from the serialization graph, but the transaction  $t_1$  continue the executing. Then  $t_1$  executes operation  $q'_1(y)$  which has to cause the conflict with the operation  $p'_2(y)$  of transaction  $t_2$ , but the transaction  $t_2$  has been purged. Thus, the graph has to contain a cycle but it is an acyclic naturally.

*Theorem. Correctness of the proposed*

The proposed protocol is correct.

*Proof:*

1. The correctness of the server side algorithm is demonstrated by the procedure of merging histories at the server.
2. The Lemma 1 and Lemma 2 prove the correctness of the client side algorithm.

## 7 Results

Further we demonstrate the advantages of the proposed data model over the classical ACID transactions for the case of application-transparent adaptation. The *Diagrams 1* and *2* represent the transactions aborts percentage as a function of specially defined X and Y parameters that determine the data sharing level.

### *X as Disjunction Depth*

X is a maximum quantity of clients that may share (be subscribed for) a fixed node.

### *Y as Disjunction Width*

Y is a maximum quantity of nodes in the disjunction of the replicas of any two clients. If Y equals to 0 the quantity of aborts equals to 0 too.

Diagram 1: Proposed Model

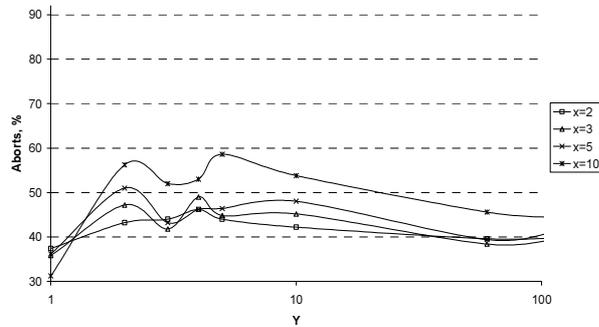
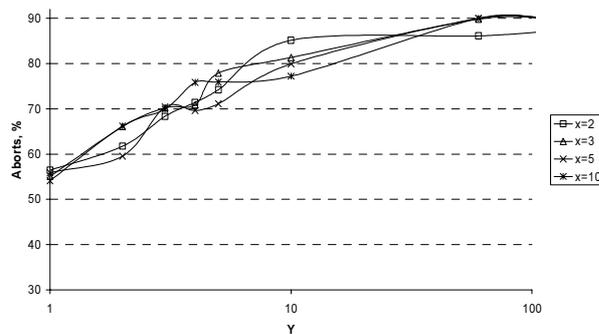


Diagram 2: ACID Transactions



## 8 Conclusion

The model of a middleware-system offered in this paper provides a model which supports a limited form of consistency in a distributed mobile environment, where classical transaction models are not suitable. This approach is suitable for typical nomadic network and may be applied to a wide class of enterprise applications working in mobile environment.

Use of special set of high-level operations is one of the most important features of the introduced solution. These operations work with the data having a tree structure with XML-styled references. The key strength of the given model, in contrast with other approaches, is that changes of the replica transfer as high-level operations and not as simple data items. In most cases, two operations permute with each other. The commutativity of those operations is based on their semantics that is defining by the target application. Thus, it reduces number of transaction aborts, and so, availability and performance of the system grow in comparison with classical transactional operation in mobile environment.

Apparently, within the framework of the offered model, two update operations being the most often operation cannot be permuted. In general, a conflict like that may be resolved only on the application layer according to its semantics. We introduce abstraction of accumulator for effective implementation of the application-aware approach in the application layer.

According to this approach, the proposed model allows application to define the policy of adaptation for each node of the data tree. Unlike simple types, an accumulator does not store an explicit value, but it contains only the base value of a simple type and the collection of operations to be applied to the base value. The operations are determined on application layer as well as the external functions that return additional information about the node of accumulator type. In our prototype system there are two different implementations of accumulator concept called list and set. According to its semantics, the application may choose the suitable implementation for each node of accumulator type.

The distinctive feature of our system is the ability of client to resolve local conflicts without communication with server. To achieve the integrity of the client's replica, its representation includes external nodes with two-way navigation. These nodes help to recognize hierarchical relationship and allow the client to resolve local conflicts. Resolving of local conflicts improves the performance of client part of application and decreases communication cost.

The protocol we use has a significant limit: it requires general local commitment of all transactions before sending of any data to server. So, short local transactions should wait completion of long ones. That issue reduces efficiency of a client.

Our future researches are focused on the issues related to replica and cache management, system scalability and system performance analysis. Also we plan to develop the strict theory of accumulators.

### *Acknowledgements*

*We thank Katja Perminova from SmartPhoneLabs, LLC for the comments which helped improve this article.*

## References

- [1] P. Bernstein, V. Hadzilocs and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, Mass., 1987.
- [2] L. Capra, W. Emmerich, and C. Mascolo. *Middleware for Mobile Computing: Awareness vs. Transparency (position paper)*. In *Int. 8th Workshop on Hot Topics in Operating Systems*, May 2001.

- [3] Il Young Chung, Bharat Bhargava, Malika Mahoui, Leszek Lilien Autonomous Transaction Processing Using Data Dependency in Mobile Environments, In: Proc. of *The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'03)* p. 138, San Juan, Puerto Rico, 2003.
- [4] N. Davies, S. P. Wade, A. Friday and G. S. Blair, Limbo: A Tuple Space Based Platform for Adaptive Mobile Application, Proceedings of the *International Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP '97)*, Toronto, Canada, 27-30 May 1997, pp291-302.
- [5] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M.M. Theimer, and B. Welch. The bayou architecture: Support for data sharing among mobile users. In Proceedings *Workshop on Mobile Computing Systems and Applications*. IEEE, December 1994.
- [6] W. Emmerich Engineering Distributed Systems, John Wiley & Sons, 2000
- [7] E. Freeman, S. Hupfer, and K. Arnold. JavaSpaces, Principles, Patterns, and Practice. Addison Wesley, 1999.
- [8] J. N. Gray, P. Helland, P. O'Neil and D. Shasha. The Dangers of Replication and a Solution. In *Conference on Management of Data*, pp. 173-182, Canada, June 1996.
- [9] IBM, <http://almaden.ibm.com/cs/TSpaces>.
- [10] T. Imielinski and B. R. Badrinath, Mobile wireless computing: Challenges in data management, *Communications of the ACM*, Vol. 37, No. 10, October 1994, pp. 19--28.
- [11] D. Kochnev Features of mobile applications development, *Mobile Communications/Russian Edition*, No. 10, 2002 (in Russian).
- [12] D.S. Kochnev, A.A. Terekhov Surviving Java for Mobile, *IEEE Pervasive Computing*, Vol. 2, No. 2, Apr-Jun 2003. P. 90-95
- [13] A. Le Hors, P. Le Hégarret, L. Wood, G. Nicol, J. Robie, M. Champion, S. Byrne Document Object Model (DOM) Level 3 Core Specification W3C Candidate Recommendation 07 November 2003. <http://www.w3.org/TR/2003/CR-DOM-Level-3-Core-20031107>.
- [14] S. Madria and B. Bhargava A Transaction Model for Improving Data Availability in Mobile Computing, in *Distributed and Parallel Databases*, 10(2), 2001.
- [15] Cecilia Mascolo, Licia Capra and Wolfgang Emmerich Mobile Computing Middleware, <http://citeseer.nj.nec.com/596660.html>
- [16] C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich. XMIDDLE: A Data-Sharing Middleware for Mobile Computing. Int. *Journal on Personal and Wireless Communications*, April 2002.
- [17] Amy L. Murphy, Gian Pietro Picco, and Gruiacatalin Roman. Lime: A Middleware for Physical and Logical Mobility. In Proceedings of *the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, May 2001.
- [18] B. Novikov and O. Proskurnin. Towards collaborative video authoring. Proc. of *the ADBIS'2003*, 370-384, Dresden, Germany, 2003.
- [19] E. Pitoura and B. Bhargava. Data Consistency in Intermittently Connected Distributed Systems. In *Transactions on Knowledge and Data Engineering*, Nov. 1999.
- [20] E. Pitoura and P. Chrysanthis, Scalable Processing of Read-Only Transactions in Broadcast Push, *IEEE International Conference on Distributed Computing Systems*, Austin, 1999
- [21] M. Satyanarayanan, Fundamental Challenges in Mobile Computing, *15th ACM Symposium on Principles of Distributed Computing '96*, Philadelphia, PA, USA, pp. 1-7, May 1996
- [22] M. Satyanarayanan, Mobile information access, *IEEE Personal Communications*, vol.3, no.1, pp. 2633, 1996.
- [23] Satyanarayanan, M, Kistler, J. J., Kumar, et. al., Coda: a Highly available File System for a Distributed Workstation Environment, *IEEE Trans. on Computers*, 39(4): 447-459, 1990.
- [24] P. Serrano-Alvarado, C. L. Roncancio et M. Adiba "Mobile Transaction Supports for DBMS" In: Proc. of *17<sup>èmes</sup> Journées Bases de Données Avancées (BDA'2001)*, Agadir, Maroc, 2001.
- [25] J. Shanmugasundaram, et. al. Efficient Concurrency Control for Broadcast Environments Univ. of Massachusetts Technical Report 1999.
- [26] G. Weikum, H.-J. Schek, Concepts and Applications of Multilevel Transactions and Open Nested Transactions, In *Database Transaction Models for Advanced Applications*, ed. A.K. Elmagarmid, Morgan Kaufmann, 1992.
- [27] G. Weikum and G. Vossen. Transactional Information Systems - Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann Publishers, 2002.
- [28] F. Yergeau, T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler Extensible Markup Language (XML) 1.0 (Third Edition) W3C Recommendation, 4th February 2004. <http://www.w3.org/TR/2004/REC-xml-20040204>

---

\* This research was partially supported by RFBR grant. No. 04-01-00173.