MSc Distributed Computing Systems Engineering

Department of Electronic & Computer Engineering

Brunel University

Master's Dissertation

Conflict detection and solution of data replications in mobile client server systems

Student: Gianfranco Clemente
Stud. Nr: 0228470

Supervisor: Dr. Thomas Owens

December/2003

A Dissertation submitted in compliance with the requirements for the degree of Master of Science

Acknowledgements

I would like to express my gratitude to Dr. Thomas Owens from the Bunel University for his guidance during this Master's Dissertation. I would like to thank the sLAB corporation, in particular Diplom Informatiker Peter Rudolph for his support and advice while working on this dissertation. This Master's dissertation would have been difficult to complete without the tremendous patience of my girlfriend and partner Rita and her willingness to support me wherever she can. I would also like to mention Mrs. Lesley Powers from the Brunel University for her support during the MSc course. She immediately went out of her way to help me wherever she could. Special thanks go out to my mother Brigitte and my father Vincenzo for showing me treasures in life that no one can buy with money.

Content:

List of Figures:

1. Introduction

Recent developments in mobile devices and mobile applications enable the user to access data on wireless mobile devices without being bound to a fixed location. Client mobile applications running on wireless devices are usually constricted by limited processing power, smaller displays and frequent disconnections to a centralized data store accessible by other clients as well. Since connections to a centralized data store or back-office system might be expensive or not always possible, data needed for the mobile client application could be replicated to the mobile device for offline manipulation. Once the connection is re-established, all offline-performed changes must be re-integrated into the back-office system. Since more than one mobile applications could work on the same set of replicated data concurrently, re-integration of such changes with the centralized back-office system could be a challenging task. Several strategies might be used at the server / back-office side to minimize or to detect conflicting operations carried out by mobile devices. If conflicts cannot be avoided some form of intelligent conflict resolution could be applied to guarantee that the data at the back-office side remains in a consistent state once the re-integration process of the locally manipulated data is started. The data replicated to the mobile device is usually manipulated by a sequence of local operations carried out by the mobile device in a disconnected mode. The sequence of local operations performed by the mobile application can be regarded as a local long-term transaction that satisfies the ACID (see [1]) properties within the client's environment. The aim of this Master's dissertation is to evaluate and to implement strategies to detect and to resolve conflicts of local client operations performed on replicated data once the data changes are migrated from the different mobile clients to the centralized global back-office system. A typical scenario where such a conflict might arise would be the scheduling of an appointment by a secretary for a manager and the manager himself. Let us assume that both persons are using mobile devices in a disconnected mode to schedule appointments for the next day. Both have received the available free time slots for the next day on their mobile devices from the server before they disconnected and are using the available time slots to schedule important meetings for the manager. Both persons will replicate their data back to the centralized back-office system once reconnected again. There is a high possibility that both persons performed conflicting operations on their mobile device that should be detected and resolved by the centralized back-office system. If automatic conflict resolution is not possible the server must generate a message and inform the mobile client that human intervention is necessary. The following chart illustrates this scenario:

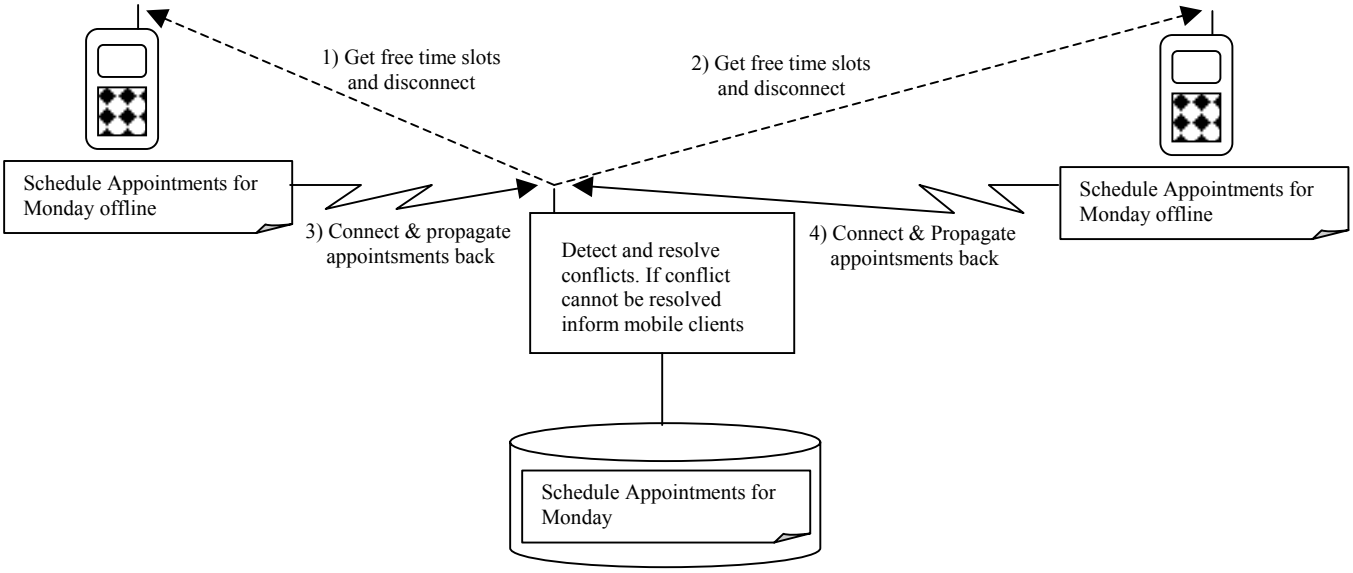


Figure 1: Offline Scheduling of Appointments

1.1 Benefits of this Master's dissertation

The concurrent access of different mobile clients to shared data items in a mobile environment along with associated conflicts that could occur, is a frequent problem that should not be dealt with separately in every software development project. To speed up the development of mobile applications and to minimize the costs, a solution in form of a general conflict detection and resolution system is presented in this dissertation.

1.2 Structure of this document

The remaining document is organized as follows. In chapter 1.3 and 1.4 the objectives of this Master's Dissertation are outlined and an initial time plan is being presented with the estimated time needed to accomplish the objectives. In chapter 2 a literature survey conducted for the interim report is presented. Chapter 3 outlines the aproach selected and starting points. In chapter 4 a mobile application framework and a useful application scenario are analyzed in order to integrate the Master dissertation findings into the mobile application framework and to test them in the context of the analyzed application scenario. The conflict detection and resolution unit's design and implementation are presented in chapter 5. Chapter 6 describes the test environment used to test the conflict detection and resolution unit and what kind of backend storage system is applied to store application relevant data. Chapter 7 provides the conclusions drawn from the presented work.

1.3 Aims and Objectives

1.3.1 Main Aim

The aim of this Master's dissertation is the development of a conflict detection and resolution module involving long-term local offline transactions.

1.3.2 Objectives

The following key activities were identified to successfully accomplish the Master's dissertation at the beginning of the dissertation:

- Code and design review of already implemented parts of the mobile application framework, into which the master dissertation findings have to be integrated.
- Review of XML-Related technologies, in particular XML-Schemas since the conflict resolution and detection module has to be configurable. XML-Schemas will be used to specify the syntax of the conflict resolving rules in XML.
- Select and analyze a typical useful application scenario that can be applied to a potential prototype.
- Evaluation of a suitable back-office systems and analyze the typical constraints of such a system, in particular any provided API.
- Development of reliable conflict detection and conflict resolution algorithms that are possibly based on existing algorithms.
- Design and implementation of the conflict detection / resolution modules. The software design is largely dependent on the conflict detection and conflict resolution algorithms to be used.
- Functional testing of the conflict detection / resolution modules in combination with a selected back-office system and the mobile application framework
- Possible adaptation of code and design depending on the outcome of the tests

1.4 Time Plan

The time plan outlined below estimated the time for the key activities listed in section 1.3.2 at the start of the Master's dissertation:

Task	Start	Stop
Code / design review	08.09.	12.09.
Review of XML-Schemas	15.09.	15.09.
Select and analyze application scenario	16.09.	18.09.
Evaluation of backoffice system	19.09.	23.09.
Development of conflict detection and resolution algorithms	24.09.	09.10.
Design & implementation of conflict detection/resolution modules	10.10.	12.11.
Functional testing of conflict detection and resolution module	13.11.	14.11.
Completion of documentation	17.11.	04.12.

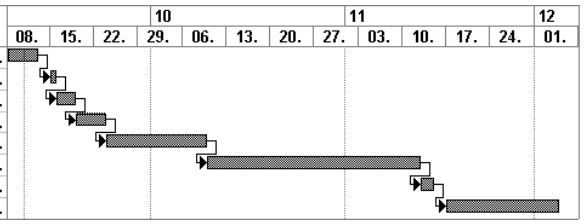


Figure 2: Gantt chart

2. Background to the project and survey

Several approaches can be taken to detecting and resolving conflicting offline operations on replicated data once re-integration of the data starts. The following section will briefly outline and discuss several approaches based on survey material. Different solutions could be used to process replicated data on mobile devices.

2.1 Replication using a pessimistic synchronization strategy

Replication systems applying a pessimistic synchronization strategy to re-integrate changed offline data into the back-office system use many of the principals applied in traditional database transactions to enforce the ACID properties and to guarantee data consistency at all times (see [1]). Pessimistic synchronization is based on locking shared data at the centralized back-office system when the data is replicated to the mobile host for offline manipulation. This approach is not suitable in a disconnected environment with many mobile clients, since one client could theoretically exclusively lock a shared data item and remain disconnected for a period of hours or even days and would release the lock only when being reconnected to the server system. This would prevent other clients from updating or even reading needed data in case of an exclusive write lock placed by one client on a shared data item and would certainly reduce the overall throughput to an unacceptable level. However the pessimistic approach could synchronize multiple clients and prevent conflicting operations. The following figure illustrates such a scheme.

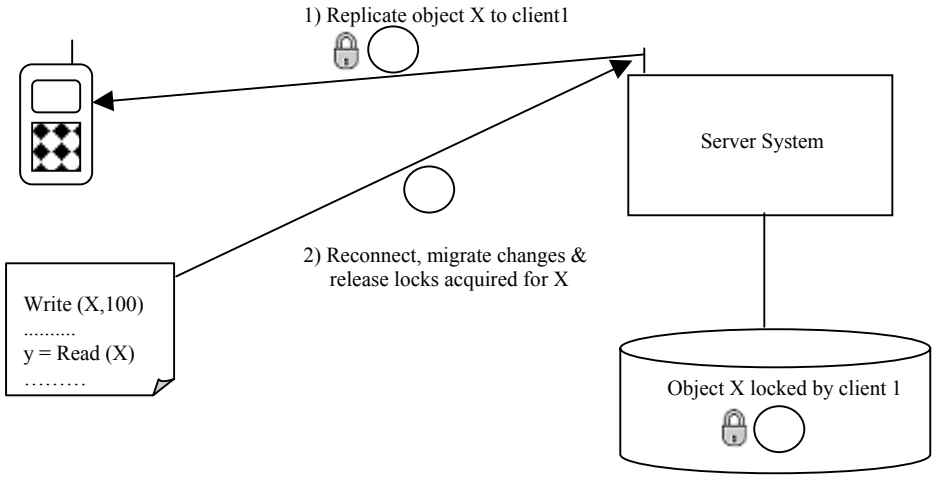


Figure 3: Pessimistic replication scheme

Figure 3 shows how object X is replicated to the mobile device after an exclusive lock was obtained at the server side. The lock is released if the client migrates the changed object X back into the server system.

2.2 Replication using an optimistic synchronization strategy

Replication systems using an optimistic synchronization strategy are relaxing data consistency requirements and use no locks at the server side during offline operations. Data can be replicated to multiple clients and each client performs a sequence of changes on the replicated data without synchronizing its activity with other mobile units. One optimistic approach is to regard the client's activity as a long-term transaction within the client's environments and to record all activities in a local log file. Each client is completely autonomous and is responsible for providing a mechanism to comply with the ACID properties of a transaction within its local environment. Since other clients could perform conflicting statements within their local transactions, conflict resolving and detection has to be provided by the server system. At the client side, local transactions are of a tentative nature and are tentatively committed at the local mobile host. Upon reconnection the transaction log is propagated back to server and the transaction is replayed. While in replaying mode locks are granted to the client and held to prevent concurrent update on shared data items by multiple client transactions replayed at the server. If conflicting operations cannot be resolved, the local transaction will be aborted. If the transaction can be replayed without violating data consistency at the server side, the changes are permanently committed by the server system. This approach is also known as a lazy or delayed commit.

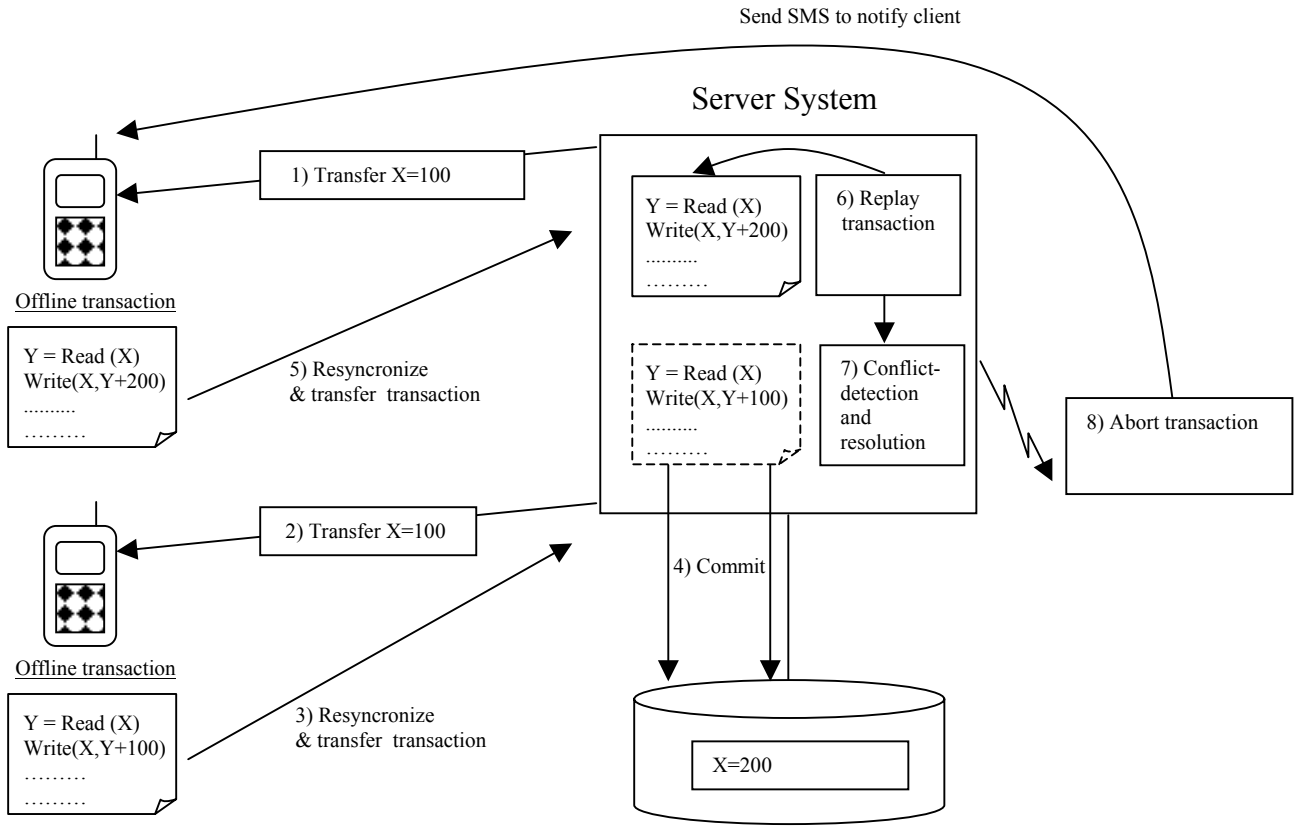


Figure 4: Optimistic replication

Figure 4 shows two mobile devices receiving the shared value X=100 and how they are performing a conflicting local update on the value. One transaction replayed at the server was committed and the other transaction had to be aborted due to an unresolved conflict. A number denotes the processing order.

2.2.1 The Gold Rush system

The above-described schema was used in a system called Gold Rush (see [2]) and the authors characterized the transaction flow as follows:

- Check out: “Partial replication of business data from the business server together with integrity constraints”
- Access: “Off-line transactional access with all the read/write information logged”
- Check in: “Re-integration of off-line transactional data with the main database”

- Conflict handling: “Detection and resolution of conflicts with redefined formulas”

The Gold Rush system provides the ability to process online and offline transactions of a mobile client. If a mobile client’s online transaction is being processed at the client side while connected to the back-office system, traditional database locks are held at the backend system for synchronization purposes.

2.2.1.1 Conflict detection

Conflict detection in Gold Rush is accomplished by associating a time stamp of the updated object with the associated unique object id at the server side. The time stamp is referred to as the last modified time. The client side provides two time stamps:

- The last modified time stamp
- The local commit time stamp

If a client transaction reads/receives a replicated object for the first time, the last modified time stamp of the client’s replicated object is set to the last modified time stamp of the object held at the server side. If a local transaction commits, the local commit time stamp is set. If the transaction is replayed at the server the client’s last modified time stamp of the replicated object and the server’s last modified time stamp of the server’s object are compared and if they are different the transaction can not be committed at the server side since another client must have altered the object concurrently at the server side. If the time stamps are equal and the object was modified within the clients transaction, the changes are stored at the server side and the last modified time stamp of the server’s object is set to the local commit time stamp of the client’s object.

2.2.1.2 Conflict solution

Gold Rush does currently not provide some form of intelligent conflict resolution. A conflict is handled by informing the mobile client of a transaction rejection. Later Gold Rush versions will include some intelligent conflict resolution on a class-by-class base or on the basis of a business scenario.

2.2.2 The Coda File System

The Coda File System (see [1], [5], [6] and [7]) was developed for Unix workstations, working in a distributed environment and supports offline manipulation of files, replicated by one or more servers to requesting clients. The Coda system also provides some kind of fault tolerance for services accessible within a distributed system by having a group of servers offering a replicated file volume to clients. The set of servers offering a replicated file volume form a so-called volume storage group (VSG). Clients requesting the manipulation of a file can contact a subset of the VSG to open a file for client side caching. The available subset of servers is called an available volume storage group (AVSG). The file request is handled by one of the servers of the current AVSG. The server chosen by the clients is the preferred server to communicate with, however the client is still checking other servers of the AVSG for the latest version of a file and might change the preferred server if another server is offering a later version of a file within the AVSG. If the preferred server is changed due to a later version available within the AVSG, the members of the AVSG are notified that some of them are hosting stale replicas. The membership of servers to an AVSG is of a dynamic nature and is usually dependent upon network or server failures. If the client requests the opening of file, the file is replicated to the client's environment. Disconnected operation begins if none of the servers holding replicas of the file is available during file operations. In such a case the set of AVSG is said to be empty. Since the client cached the file locally, offline manipulation can resume immediately. The client's offline operation is only suspended if a resource that is not available in the local cache has to be requested from one of the server of the AVSG group and the set of AVSG is empty. The file replication is based on Coda version vectors (CVV). The server offering a version of a particular file associates the file with a CVV. The CVV corresponds to a logical vector clock time stamp with one element for each server in the associated VSG. Each vector element corresponds to the number of changes applied to a version of a file kept at a particular server. Since the CVV is based on a logical vector clock time stamp, the following section outlines the principles of a vector clock.

Definition vector clock:

A vector clock is a logical clock where a vector V_i contains n values. Each process or node P_i in a distributed system contains a Vector V_i that represents a time stamp. The length of the vector n represents the number of processes or nodes involved in an ordering scheme. A vector element at V_i [x] of process or node P_i represents the number of events that happened

within a process or node P_x that are detected by P_i. Each process or node initializes its Vector elements to 0. If the time ordering scheme starts, the schema works as follows:

- If process or node P_i generates an event $\rightarrow V_i[i] = V_i[i] + 1$
- If process or node P_i sends another process or node a message, this also considered an event generated by P_i therefore $\rightarrow V_i[i] = V_i[i] + 1$. The vector V_i is also transmitted as part of a message to a different process or node
- The event "P_x receiving a message from P_i with an embedded Vector V_i" is reflected by P_x updating its vector to $V_x[i] = V_x[i] + 1$ and then $V_x[y] = \max(V_x[y], V_i[y])$ for all $y \in$ index of vector.

The event **"a happened before b"** could also be expressed as

Timestamp of a < timestamp of b

where the partial order $\leq, <$ is defined as followed:

Timestamp1 ≤ Timestemp2 if:

Timestamp1 [y] $\leq$ Timestemp2 [y] for all $y \in$ index of vector

Timestamp1 < Timestemp2 if:

Timestamp1 [y] $\leq$ Timestemp2 [y] and Timestamp1 $\neq$ Timestamp2
for all $y \in$ index of vector

2.2.2.1 Conflict detection in Coda

The CVV of a file represents the update history of a file version and is used to detect conflicting operations by clients on a version of a file. After a client updates a file and closes it, all servers in an AVSG are sent the updated contents of a file and the corresponding CVV by an update message. Each server updates the file version and sends a positive acknowledgment back to the client. The client increases the vector element of all servers that answered with a positive acknowledgment by one, indicating that the file has been updated by the particular server and distributes the vector to all servers of the AVSG.
A conflict is detected if for two vector v1 and v2, neither v1 $\geq$ v2 nor v2 $\geq$ v1 holds. This means a file version has been concurrently updated by two clients and could have been caused

by servers belonging to a different AVSG having received only one update by one particular client that the other server does not know about. This could very well happen if the clients are part of different AVSG reintegrating the changes with different AVSG servers because of a network disconnection between them. We could say the network is temporarily partitioned. Because of the network failure and partitioning, the client is only able to access a subset of VSG servers. This implies the other servers being not part of the client's AVSG are unable to receive the update message sent by a client to the AVSG servers. The concurrent conflicting changes applied by two different clients on the replicated version of a file residing on different AVSG server can not be detected during network interruption. The discrepancy will be discovered once the network failure disappears and one client receives in a routine check the two conflicting CVV's from the two mentioned servers that are now part of the same AVSG. Another conflict could be introduced if we assume that two clients are updating a version of a file in their local cache and both clients are using the same preferred server to reintegrate their changes. A write-write conflict will be detected if:

$$V_{Server} > V_{Client}$$

Since the client's CVV is smaller than the server's CVV, another client must have updated the file concurrently at the server side.

2.2.2.2 Conflict solution in Coda

No real intelligent conflict resolution is provided by the Coda system. Conflict resolution usually has to be handled manually.

2.2.3 Other simpler optimistic synchronization strategies

The main optimistic synchronization strategies discussed so far are using offline transactions to manipulate replicated data locally and are offering features such as local tentative rollback and tentative commit. I would also like to briefly review a much simpler protocol, currently used in Palm PDA's that is dealing with offline manipulation of replicated data, conflict detection and resolution.

Palm's HotSync Protocol

As outlined in paper [3], Palm' HotSync Protocol uses local databases for each application to store metadata that indicate the status of replicated data on the device. For each data record in

the database, a unique record identifier, the location of the data in the memory and the status flag is maintained. The status flag indicates status information such as if the record is new, deleted or changed. On synchronization with the stationary host the records, with a status flag indicating some changes, are transferred to the host. The synchronization logic depends on software units called conduits and can vary from application to application. The conduit is responsible for conflict detection and resolution. One conflict resolution strategy could be that a stationary updated record has a higher priority than the conflicting hand held record or vice versa. Other techniques to detect conflicting data changes could be the use of logical timestamps or version vectors.

2.3 Reservation approach

The reservation schema in the context of mobile transactions is a combination of the pessimistic and optimistic synchronization strategy used in replicating systems. A mobile client could request data from the server to be replicated to the local mobile environment and could request different kinds of guarantees that are granted by the server for a specific period of time. Once the lease expires the guarantees made by the server associated with some data set are not valid anymore.

2.3.1 The Mobisnap system

The Mobisnap system (see [4], [8]) is an implementation that uses mobile client transactions, specified in a PL/SQL dialect, to manipulate partially replicated data on a mobile device. The language supports the specification of pre and post conditions. The client system maintains two versions of replicated data if enough resources are available. One version represents the server side committed data and the other version represents a tentative copy used to perform local tentative transaction. If the resources available at the mobile client side do not allow two versions to coexist just the tentative copy will be provided at the client side. The local tentative copy could be regarded as the future state of the central database once the client transaction is successfully replayed at the server side. The provided API can access both client database versions. In case of a transaction conflict that is irresolvable at the centralized database, the client transaction is aborted at the mobile device. Up to this point the approach is similar to the one discussed in section 2.2, however the mobile client can additionally use a reservation mechanism. The reservation mechanism provides guarantees made by the server as mentioned above. The different reservations provided by the server for a period of time are:

- Escrow

  The data will be partitioned and replicated to the client so that partition p1 is assigned to client c1 and partition p2 is assigned to client c2 where p1, p2 $\in$ data and p1 $\cap$ p2 = {}. The client c1 is exclusively using p1 and c2 is exclusively using p2
- Slot

  Server guarantees the right to insert a record
- Value-change

  The exclusive right to change some values in the database is granted
- Value-use

  The right to use a given value of a database field even if the value was actually updated in the meantime.

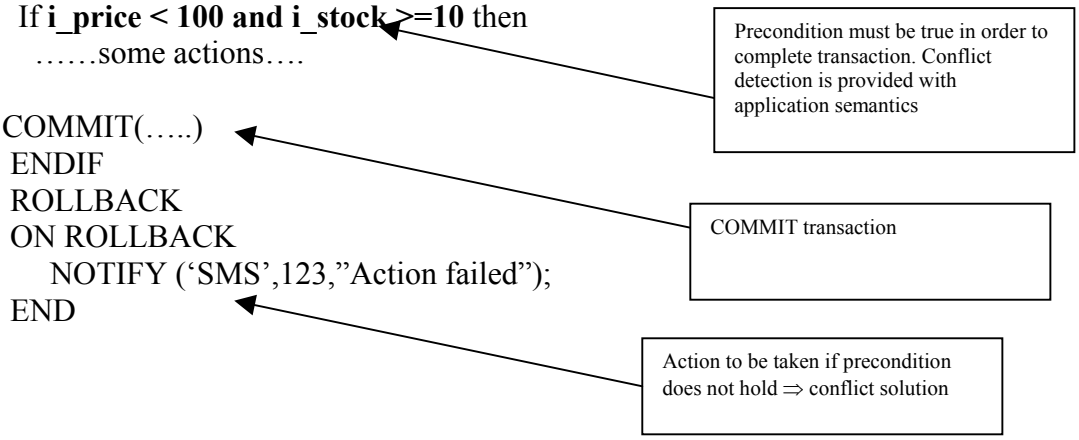
The provided guarantees enables the client to proceed and perform a local tentative commit in such a way that if the transaction depends solely on values guaranteed by reservations, the local transactions is guaranteed to be committed at the server side if the reconnection is established before the leases expire. It should also be noted that some of the reservations types are using traditional database locks held at the server during disconnection. The type of reservations and some conditions have to be assigned to an administrator specified table.

2.3.2 Conflict detection and resolution in Mobisnap

In traditional transaction systems, read-write and write-write conflicts are detected or avoided. The Mobisnap system uses an additional approach. Mobisnap takes application semantics into account to detect and resolve conflicting operations. A traditional database system, using optimistic synchronization mechanisms to synchronize concurrent operations, might only allow transactions to proceed if shared data was not concurrently updated by other users. In Mobisnap pre and post conditions can be specified in a PL/SQL language. In case pre or post conditions are violated, alternative operations can be supplied and executed. The pre and post conditions are evaluated at the server side when the transaction is replayed. The pre and post conditions are embedded into PL/SQL statements received from the mobile client. The following lines represent a small PL/SQL script with a conflict detection rule:

```
DECLARE
  i_stock INTEGER
  i_price  FLOAT
BEGIN
  SELECT stock, price into i_stock , i_price from product where ID=80
  If i_price < 100 and i_stock >=10 then
    ……some actions….

 COMMIT(…..)
  ENDIF
  ROLLBACK
  ON ROLLBACK
     NOTIFY ('SMS',123,"Action failed");
  END
```



By taking application semantics into account, multiple clients can concurrently perform conflicting operations on shared data if no exclusive reservations are acquired on the shared data item and still be accepted by the central database server, as long as some semantic conflict rules are not violated. Let us assume that two sales persons are working out in the field selling computer devices. As long as the total number of items in stock does not reach a negative value, the central database server could accept both mobile client transactions tentatively committed at the mobile device and containing conflicting operations.

2.4 Conflict avoiding vs. conflict associated replication synchronization strategies

The replication synchronization strategies could be grouped into two categories:

1. Conflict avoiding replication synchronization strategies

This type of replication synchronization strategy is usually using locks to prevent concurrent conflicting operations by mobile clients. Other mechanism could be applied as well such as versioning of concurrent updated shared data. If two mobile clients are issuing conflicting operations on a shared data item, two new versions of the changed items could be reintegrated into the server system. This mechanism is also applied at version control systems used in the software development community. If conflict

avoidance with the help of locks is used, the degree of concurrent data processing is reduced since a particular mobile device might exclusively lock some shared data items.

2. Conflict associated replication synchronization strategies

No locking mechanisms are applied in "conflict associated synchronization strategies", therefore giving each mobile client maximum authority over the replicated data. Compared with "conflict avoiding synchronization strategies", global consistency cannot be guaranteed for a mobile client at all times, since each client could theoretically operate on data items already updated by another mobile device at the server side. On the other hand availability of data items is increasing by allowing long term disconnected mobile clients to update local available data items concurrently. Since all mobile units allow conflicting operations, algorithms to detect and resolve conflicts upon synchronization with the server are needed to guarantee consistency of the centralized database system. Updated data items, residing at the server, are usually downloaded to the client's environment in the synchronization process to decrease the data divergence of mobile devices. If divergence of local data or the number of mobile increases, the frequency of transactions aborting at a local level, due to unresolved data conflicts discovered at the server increases if a transactional approach is followed.

2.4.1 Conclusions on conflict handling

Conflict handling in "conflict associated synchronization strategies" could be characterized as data oriented and operation oriented.

2.4.1.1 Data oriented conflict handling

In data oriented conflict handling systems the dataset of the client, obtained at the time of last synchronization, is compared with the current corresponding dataset of the server at the time of resynchronization. If a mobile client updated replicated data and the local data set state obtained at the time of the last synchronization, does not conform to the current centralized database state, a concurrent update of the server by another client took place and conflict resolution has to be activated at the server side. The data oriented conflict handling has the major disadvantage that all operation semantics of local mobile client changes are not preserved and therefore cannot be used in conflict resolution algorithms.

2.4.1.2 Operation oriented conflict handling

Operation oriented conflict handling could use timestamps or as in data oriented conflict handling the pre-image of the client data state before the clients data diverged from the server data and compare it to the image of the servers data state upon synchronization time to detect concurrent client changes by different mobile devices. All operations performed by the client could be held in a protocol file. The operation history is saved and the semantics of the changes are not lost as in a data oriented conflict handling. The change history and type of operations performed could be successfully reused in any intelligent conflict resolution schemes as proposed in chapter 2.3.2

3. Approach selected, aspects considered and starting points

3.1 Approach selected

The “operation oriented conflict handling” approach seems to be the most promising path to follow. By taking operation/application semantics into account, a much more flexible conflict resolution is possible.

Some conflicts caused by multiple clients accessing global data can be ignored when taking operation/application semantics into account (see chapter “2.3.2 Conflict detection and resolution in Mobisnap”). An optimistic synchronization mechanism, involving the replay of local transactions at the server side, guarantees maximum autonomy of local clients since clients can access and manipulate data even without being connected to the central data storage system. Possible temporarily data inconsistency is acceptable for some mobile applications if they are operating on mostly partitioned data sets and the majority of all data accesses are of a “read only” nature. The optimistic synchronization approach would certainly be problematic if a large number of mobile client are changing a small set of global data frequently. This would result in large number of conflicts that could influence the overall acceptance of the application by the customer especially if automatic conflict resolving cannot be performed.

Pessimistic synchronization methods involving locks, obtained by the client at the time of being online with the server, are not considered at all, since it would decrease availability of data units. (See chapter 2.1 Replication using a pessimistic synchronization strategy)

3.2 Aspects considered and starting points

The work presented in this dissertation is restricted to conflict detection and resolution at the central server system. The conflict detection and resolution modules implemented by this Master’s dissertation are meant to be part of a mobile application framework developed by sLAB corporation. The existing framework provides the infrastructure such as the recording of client transactions on a local mobile device and the http-communication mechanisms with the server.

The major constraint to any possible intelligent conflict detection and resolving is the modular integration of existing back-office systems without modifying the algorithms or major components of the implemented software developed by this Master’s dissertation. Internal modifications of existing back-office systems such as SAP/R3 might not be possible. This

should be considered in the design of the conflict detection and resolution modules. The conflict detection and resolution sub system, processing local mobile transactions at the server side, might be starting and using back-office system transactions to accomplish the business objectives. A back-office transaction could be a database transaction of a database system that is part of the back office system. The database transaction could be triggered by a local mobile transaction.

The operations performed on a local offline client are actually operations replayed and run against an existing back-office application-programming interface used to manipulate back-office data. Conflict resolving strategies used by the system must be configurable by adapting an XML-configuration file(s). The following chart illustrates typical system configuration:

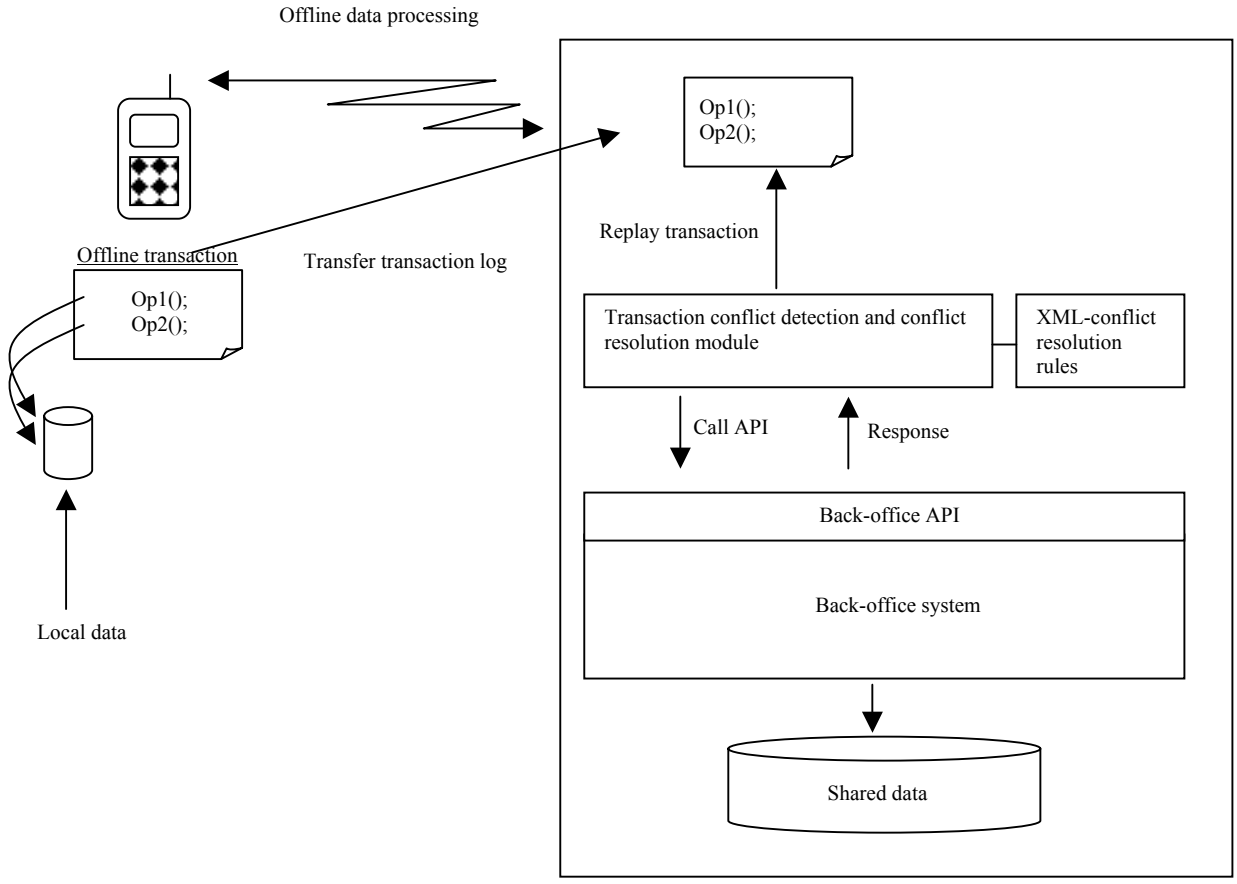


Figure 5: System layout

4. Framework analysis / Application Scenario

The dtf/AF Framework was developed to speed up the development process of mobile applications that can be executed on different mobile devices, ranging from cellular phones to small handheld computers.

The framework supports mobile offline transactions, the replay of mobile offline transactions at the server side, modular model-view-controller architecture and the XML creation of client side GUI's. The framework is written in Java, a platform independent language, known for its clean and simple object oriented language structure. Server data is replicated to the mobile client's environment for offline data manipulation. An analysis of the major components of the framework is necessary in order to integrate code produced by the Master's dissertation so that it can be proven that the conflict detection and resolution unit actually works in combination with the application framework when offline manipulated data is transferred back to the server system. Of special interest is the internal data structure of the framework since the conflict detection and resolving process has to interact with the internal data structure already in place. Modifications of existing framework code is necessary to produce a clean interface where the conflict detection and resolving modules could be plugged in.

Any necessary modifications of the framework are part of the development process of the conflict detection and resolving unit therefore an inside data structure analysis of the framework is, as already mentioned, inevitable.

4.1 Application Scenario

The *sLAB* corporation and the *Frauenhofer Institute Germany* have developed a mobile client application called *Teleservice* for a service technician, maintaining machines out in the field for a customer. The mobile application framework was used to develop the *Teleservice* application. The application created is used in the conflict detection and resolution process and serves as an example to explain the internal structure of applications that are developed with the framework. The technician using the Teleservice application will update a mobile device with the daily service emergency requests received by a dispatcher of the service headquarters. The technician will drive to the customers with his list of repair requests on his mobile device and will hopefully solve all technical defects. The technician is able to edit or create a new repair request on his mobile device while being offline or fetch new emergency repair requests from the service headquarters while being out in the field. The service

technician could create a report about the service performed and record all spare parts used while servicing the customer. After all work is completed he will replicate his local data to the service headquarter to inform them about the service incident and his new contingency of spare parts left for further emergency repair requests. The service dispatcher, knowing about the technician's contingency of spare parts, could order new spare parts from his supplier if necessary and would be able to judge if the technician is still able to fulfill particular repair request. The following figure displays the main dialog of the application:

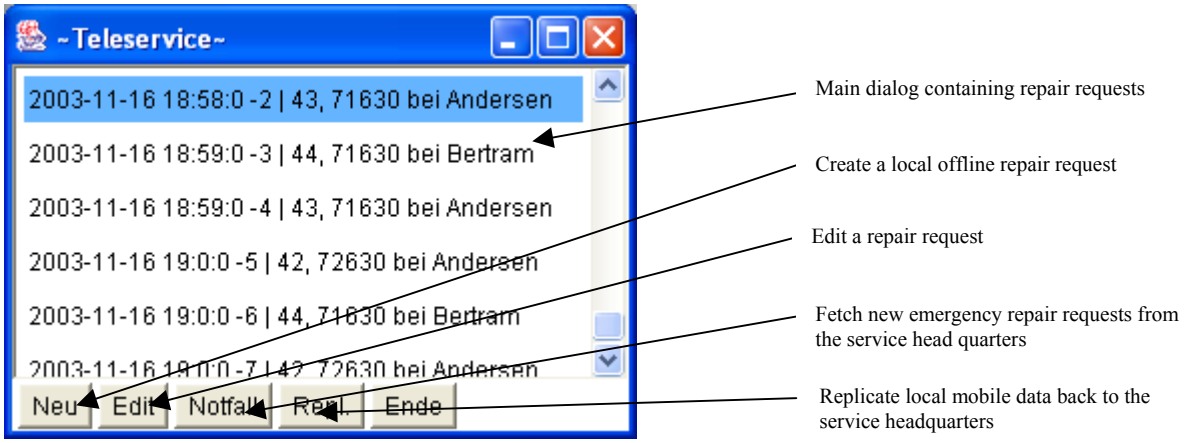


Figure 6: Teleservice main dialog

Figure 6 shows a list of emergency requests separated by a date and details of the service location. The service technician could select a repair request to edit the request, create a new request in case the synchronized repair requests do not match new requirements out in the field, fetch new requests from the headquarters or synchronize local data back to the headquarters.

4.1.1 Observer pattern

All data items of an application created with the framework are organized as subjects in an observer design pattern. The observer pattern uses different objects called observer and subject to model a certain dependency between those two objects. An observer's consistent state depends on the current state of a subject. We could say an observer depends on the current data of the subject. An observer needs to be notified when the state of the subject changes in order to be able to react to the change of state by transferring itself into a new consistent state.

A variation of the observer pattern is the MVC (Model-View-Controller) pattern where the model represents the subject, the view represents the observer, which is typically a graphical representation of the model and an additional component called controller reacts to events triggered by a GUI that changes the model. For simplicity I will refer from now on to the subject as the model.

In the mobile application framework all data items are models that notify dependent objects of any changes of state. The main advantage of this approach is the clean separation of data and dependent objects because it reduces the overall coupling of the software components. The model could have for example many graphical observers also referred to as views that update their graphical representation of the model once the model changes. Additional views could even be plugged in during runtime of the software system. The following figure illustrates such a design. The number denotes the order of the activity:

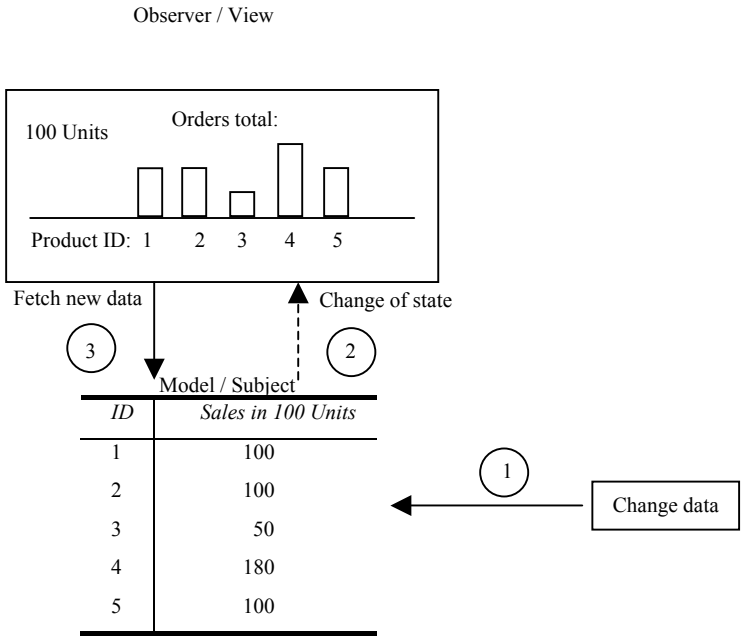


Figure 7: Observer pattern

4.1.2 Tree structure

All models of a mobile application created by the framework are ordered in a tree like structure. A tree structure consists of nodes where one node is connected to another node and the connection is called an edge. A sequence of nodes connected by edges is called a path. There can only be one path from one node to another node in a tree structure otherwise it would have to be a graph. A node could be the root node of one or more sub node. The root

node is called the parent node and the sub nodes are called child nodes. Nodes at the same level of a tree are referred to as siblings and nodes without children are called leaf nodes.

A unique ID, that is part of the model's data, represents the object identity within a set of models of a certain model type in the framework. The ID could be compared to a primary key used in relational database systems to uniquely identify a data row within a table. The reason why a tree structure was chosen in the application framework is because the model-objects are automatically built from an XML document and the structure of an XML-document resembles the structure of a tree. The organization of the Teleservice models is shows in figure 7.0:

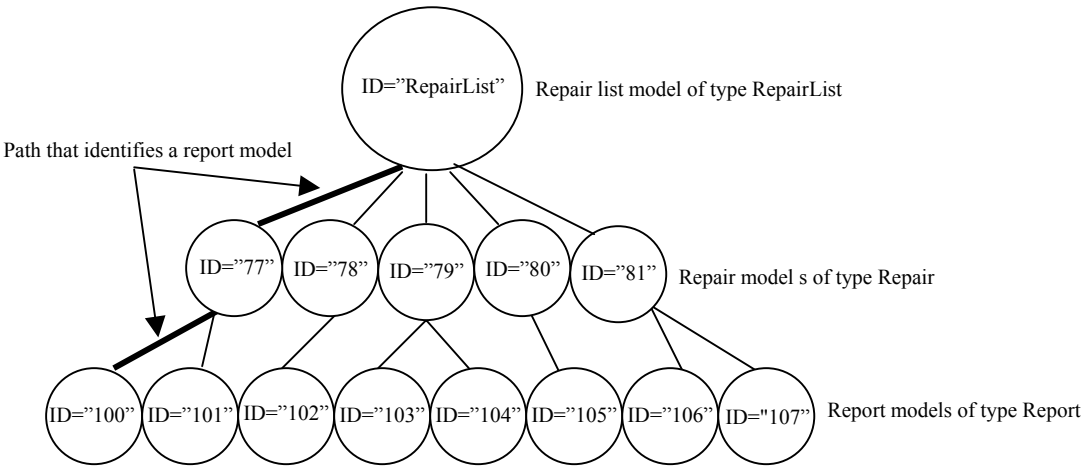


Figure 8: Internal data representation of the prototype application

Figure 8 illustrates how service requests are organized in memory. The service requests are internally encapsulated as *Repair*-model objects. A *RepairList* -model is a container for a collection of *Repair*-models. Each *Repair*-model could contain several *Report*-models. As the figure indicates, identifying a particular model within a tree is possible by describing a path from one node to another node. I will use the following terminology from now on to describe a path:

/Type of node:ID=”X”/Type of sub-node:ID=”X”

Example:

Identifying a report model with id=100 of a repair model with id=77 that is part of a repair list model would be: /RepairList:ID=”RepairList”/Repair:ID=”77”/Report:ID=”100”

The type is required since the id is only unique within a set of certain types of models. In other words all model-IDs are only unique among the siblings of the tree whereas the combination of model-ID and type identifies a model globally.

Figure 6, displayed above, is the graphical representation of the *RepairList* container. The following two dialogs represent the graphical representation of a *Repair* model and the associated *Report* model.

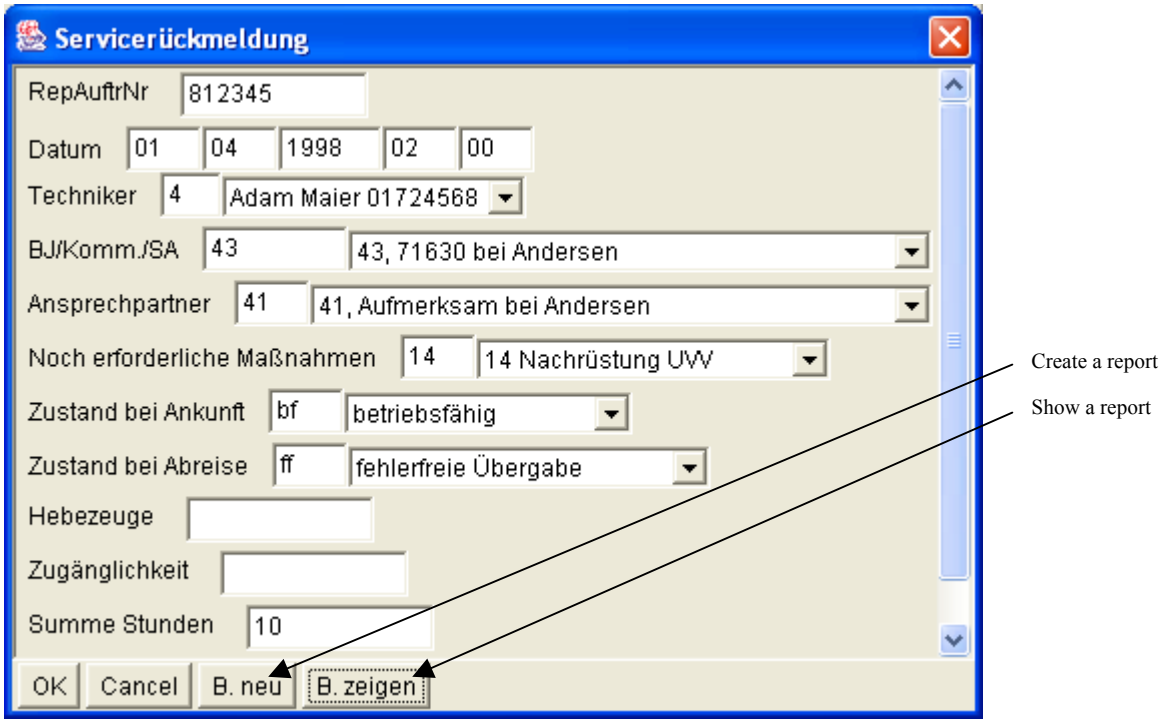


Figure 9: Teleservice edit dialog of a repair model

Figure 9 displays administrative data of a particular emergency repair request. The dialog enables the user to open a new dialog to add or edit reports of a particular service request.

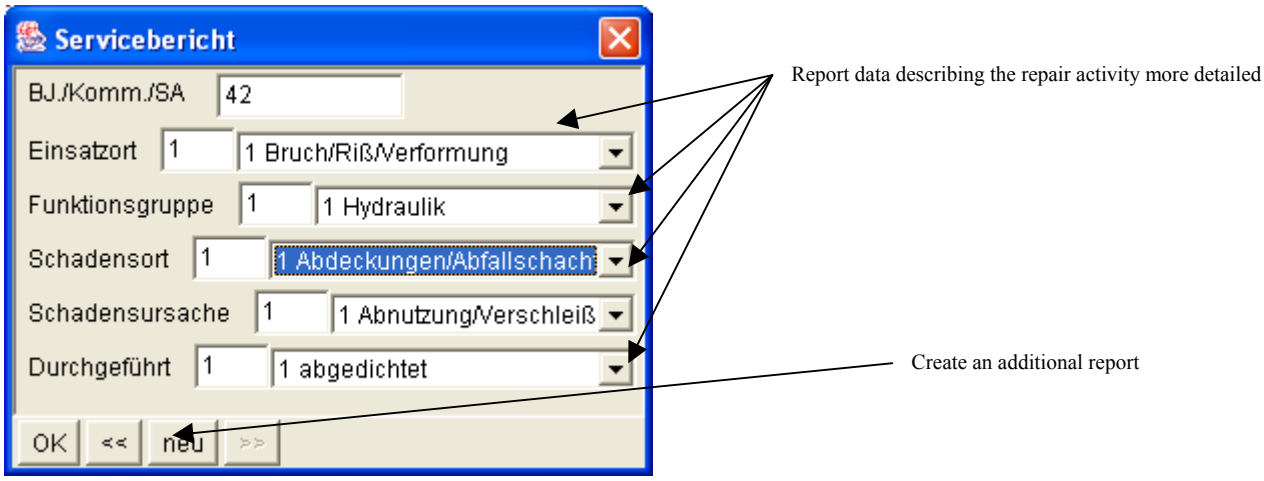


Figure 10: Teleservice edit dialog of a report model

Figure 10 represents the *Teleservice* dialog to edit or create a report belonging to a repair request.

The models involved in storing the Teleservice application's data are:

- A *RepairList* model object containing repair model objects. The *RepairList* model object acts as container for all emergency repair requests.
- *Repair* model objects containing administrative data involved in the repair process.
- A *report* model object describing the actual work accomplished.

4.1.3 XML and XSD

The associations, structure and rules about valid content of types of XML data in XML-documents are predefined by XML-schemas in the application framework. The types of XML-data can be defined and described by using a language called XSD (**X**ML **S**chema **D**efinition Language). Complex data types can be constructed out of predefined elementary data types, that are part of the W3C-XSD standard, and XSD language elements such as elements, types, attributes and others. Complex data types could even be composed out of other complex data types. Inheritance is also supported and improves reutilization of existing data types. The XSD language can express the repetition of certain types of XML-data. XML-schemas are used to verify that an XML-document, that contains actual instances of XML-schema type definitions, is syntactically correct. An XML-parser verifies the correctness of the XML-document.

In the mobile application framework XML-schemas are mainly used to automatically generate Java model classes while building the application. The XML-document is processed during runtime by the framework to build instances of the generated classes, fill them with data and organize them in a tree-like structure. Every model class is automatically generated while building the application and contains code that can be interrogated for a type description of the class during run time. This is possible since the schema type definitions are used to construct the class and type description code is embedded during the code generation phase. This is necessary because different model classes sharing common interfaces are part of the tree structure in memory and need to be asked about their type description in the conflict detection and resolving stage. The Java-Reflection mechanism that is part of the Java language cannot be used for this task since many mobile devices do provide enough computing power to allow the Java-Reflection mechanism to run efficiently.

We could summarize that the mobile application framework consists of components that accomplish the following tasks:

- Classes are generated out of XML-schema type definitions where the attributes of a class are constructed out of attributes defined in a schema-type

- Models of the generated classes are instantiated and attributes of the objects are initialized with the attribute values of XML-tags

- Objects are organized in a tree-like structure. The organization of the tree depends on the organization of the XML-tags in the XML-document

One advantage of this approach is that the complexity of creating new model data types is shielded from the developer and complex data structures can be constructed by specifying XML-documents. The underlying programming language could be even changed without modifying any XML-Schemas or documents.
The following figure describes how an XSD-Schema is used to automatically build correspondent Java classes of the Teleservice application during the application development process:

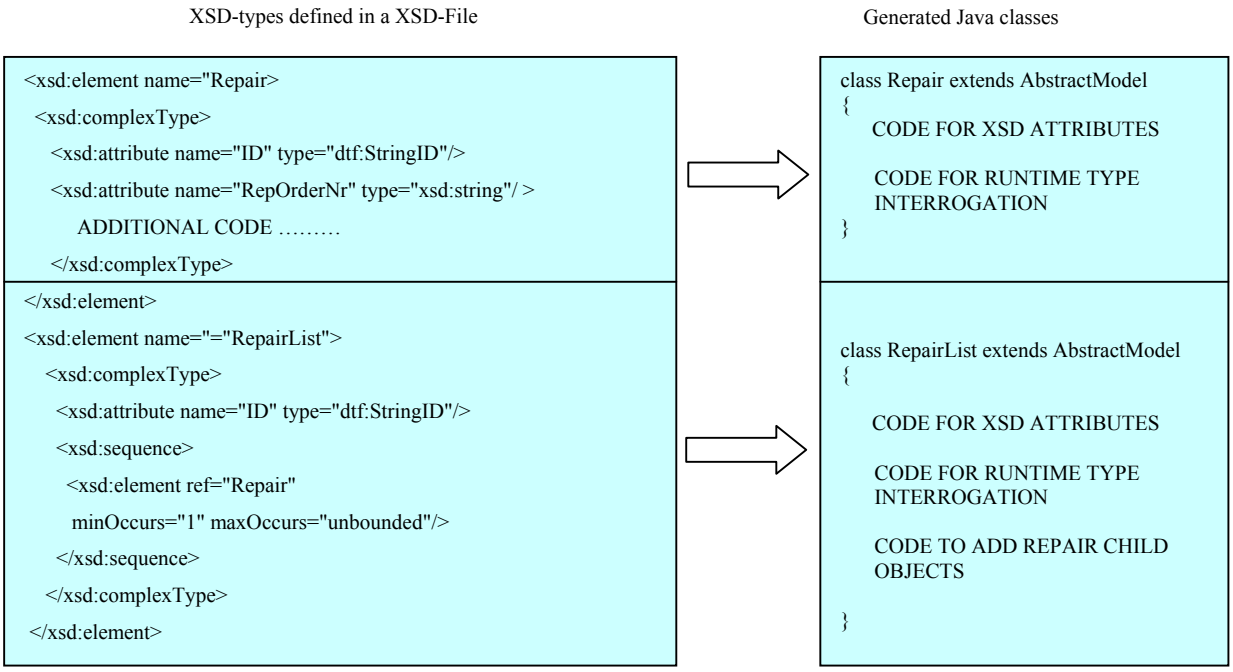


Figure 11: Automatic conversions from XSD-types to Java classes

Figure 11 shows XSD-type definitions for a *Repair* and a *RepairList* type that are used to automatically generate a *Repair* class and a *RepairList* collection class while building the application. An object of the *Repair* class could be used to store data of the repair of a machine ordered by a customer. An instance of the *RepairList* class is a parent node used to store child node *Repair*-models.

The XSD-Schema can not only be used to build corresponding classes during the development phase but could also be used to verify that a XML-document containing instance data for particular model instances of generated Java classes conforms to the structure defined by the XSD-Schema type definitions. The following figure outlines how an XML-document is parsed and the Teleservice applications data structure is constructed:

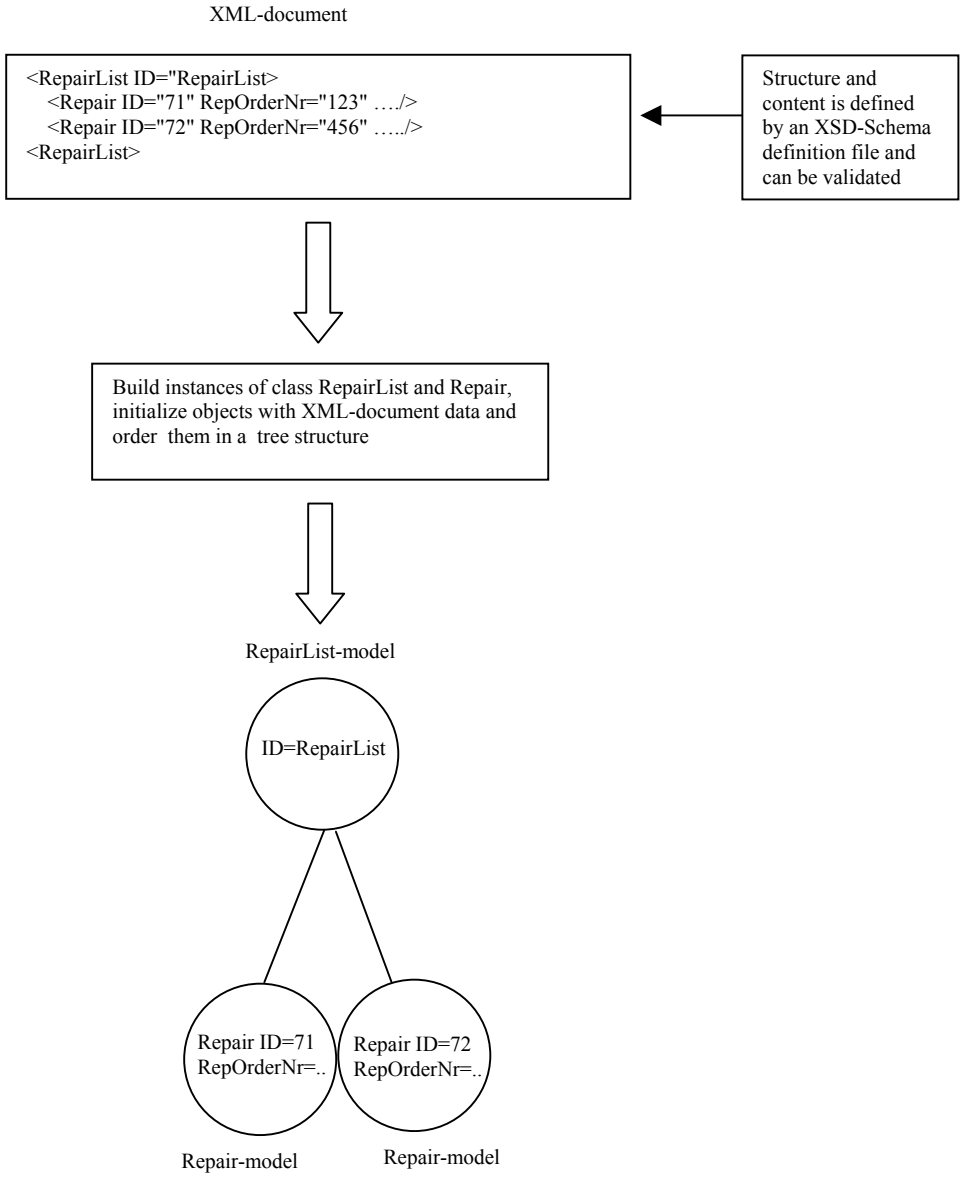


Figure 12: Processing of XML-documents

Figure 12 documents how an XML-document consisting of XML-tags and attributes can be transferred into tree structure during runtime. An XML-parser reads the XML-document conforming to a structure defined by an XSD-schema definition file. During the parsing process corresponding model class instances are constructed and a tree structure is built. The tree structure resembles the structure of the XML-document

4.2 General client interaction with models

The client infrastructure in place provides for the manipulation of models over a GUI. Events triggered by a mobile device's GUI result in the manipulation, creation or deletion of models

residing on the mobile device. All existing client models are received from the server at one time or are created newly on the mobile device. The models received by the mobile client from the server contain the same object-ID's that are also kept at the server side. Every user interaction with the model is part of a local transaction and involves predefined data access operations specified in a data access layer. The data access layer is nothing else than a special class that implements a predefined Java interface and contains data access operations to retrieve, delete, create or change stored models. A user GUI interaction selecting, changing, creating or deleting objects automatically starts a new local transaction resulting in calls of predefined data access operations identified by unique operation-IDs. The sending of data access operation IDs to the data access layer is triggered by the GUI and delivered by the framework. The data access layer provides a mapping of operation IDs to operation names. All operation names together with input and output parameters will be tracked in a transaction log file. The transaction log file will be transferred to the server at the time of the next synchronization with the server, assuming that a local commit was issued prior. The data access layer has to be adapted for every kind of mobile application. A mapping from operation-IDs to operation names has to be created for every mobile application. Data access operations could be categorized as:

- Operations that retrieve or create models
- Operations that change or delete models

1. Operations retrieving or creating models:

Data access operations that are retrieving existing models from a data storage unit are provided with a model-ID as an input parameter. The model is returned by the data access operation and a numeric value representing the retrieved model is stored as a result parameter together with the operation's name and the model-ID input parameter in a transaction log file. The transaction manager records the model reference number and other operation relevant data if the retrieved model is part of a subsequent operation input parameter list or the returned result of a subsequent operation. In case a data access method is called that creates a new model, a subsequent negative model-ID input parameter, the operation name and the numeric result parameter representing the created model are stored in the transaction log file. It should be noted that all data access operations of the application framework are actually receiving or returning model objects

and not model reference numbers. Numerical representations of the models are just used in the transaction log file as a model reference. Since all subsequent operations recorded in the transaction log are not referencing to a model directly but to a model reference number instead, retrieving the model with the same model-ID at the server side and binding it to the recorded model reference number will guarantee that subsequent operations of the transaction log file will manipulate the corresponding model at the server side during the transaction replay phase.

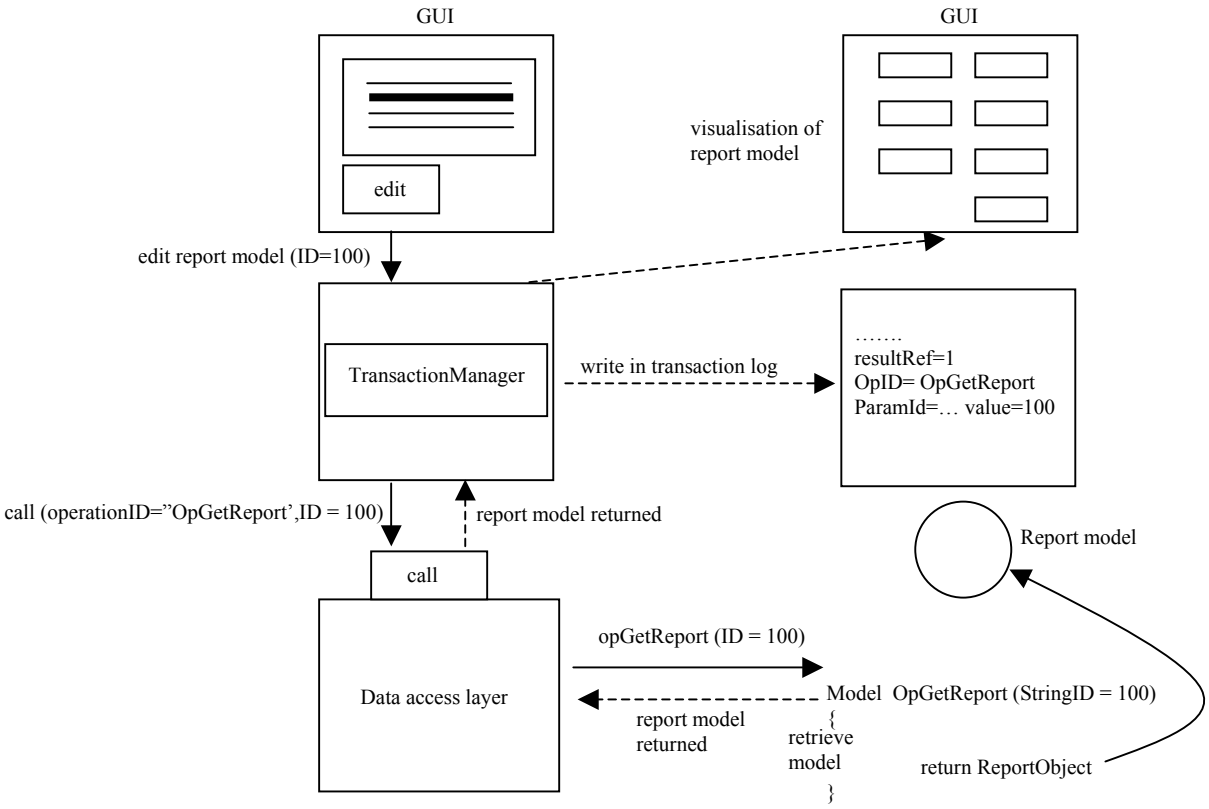


Figure 13: Retrieving a model

Figure 13 show a graphical user interface listing reports that can be altered. If the user wants to edit a report, the transaction manager will start a transaction and an interface method of the data access layer with an operation-ID “OpGetReport” is activated. The operation-ID “OpGetReport” is mapped to a method that returns a report model. The data access layer delivers the requested report and the transaction manager updates the transaction log file with the operation-ID that conforms to the operation name, the model reference number representing the model delivered and the input parameter identifying the requested object. A new dialog displays the data.

2. Operations changing or deleting models

Data access operation changing models, are receiving the model to be changes as an input parameter. The transaction manager will record the involved model reference number of the model passed as an input parameter together with the operation name. It should be noted that a model retrieve operation would always precede a model change operation. Deleting a model is straightforward by providing a model id to the appropriate data access operation and removing it from the internal data storage unit. The following figure represents parts of the transaction log file created after changing a report-model. Please note that, for reasons of simplicity not all attributes of the XML-tags used are embedded in the transaction log file.

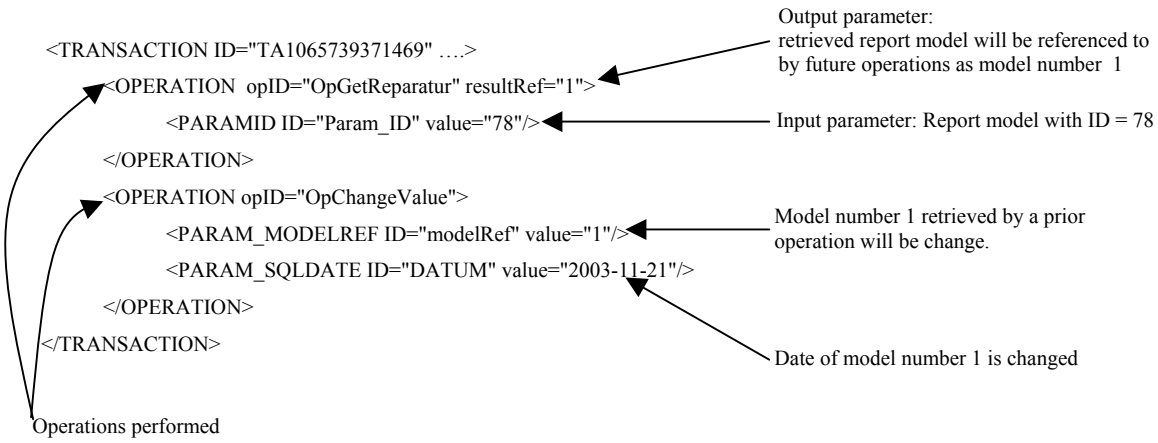


Figure 14: Transaction log file generated

4.3 Client communication with the server and back-office data processing

4.4.1 Client communication with the server

As indicated in chapter 3 local transactions are recorded and replayed at the server side. The client device uses a reliable GPRS-TCP connection to communicate with the server. The server uses a TCP-server socket connection to listen for incoming client transaction log files. The transaction logs are received and stored in a predefined directory for further server side data processing. Concurrent connections to the server by multiple clients produce multiple transaction log files. The intention is to replay the transaction logs in a FIFO order.

4.4.2 Server side model manipulation

The server side behavior of the framework concerning the manipulation of models is almost identical to the client side. The main difference is that instead of having a GUI trigger the execution of data access operations, the transaction log file is parsed and every operation recorded in the transaction log file results in a call to the corresponding data access operations in the data access layer at the server side. The server's data access layer has to be provided by the developer and acts as a wrapper to a back office system such as SAP/R3 or a plain relational database. The server's data access layer is actually the interface of the mobile application framework to the outside world. This Master's dissertation is providing an extra layer between the data access layer and the replay layer.

5. Design and implementation of the conflict detection and resolution units

5.1 Replication of model data from client to server without considering conflicts

Shared global models are being replicated from a centralized data storage system to the environment of a mobile device to increase availability of data in case the device gets disconnected. Local operations executed on a local mobile device while being offline, are transferring local models from a state n to n+1. Let us assume that global models will not be changed by any means while operations are altering local models. We also assume that the pre-image of the client models before local operation are executed on them, is identical to the corresponding server models. If we apply all operations performed on the local models also on the corresponding global server models at a later time, then the state of the global server models involved in the operations must be identical to the state of the corresponding client models. In this Master's dissertation, data replication from the client to the server system is not accomplished by transferring the models but by replaying client side operations at the server system.

5.2 Replication of model data from server to client

Since all clients need to be provided with a copy of the server models at one time the following procedures are used to replicate server models to a mobile environment:

1. If the client application is run for the first time an empty transaction log file is transmitted to the server. The server system replicates a copy of every model to the mobile device.
2. If conflicts are detected that could not be resolved automatically while replaying an operation of the transaction log at the server, the client can receive a current copy of the server models.

5.3 Design of the conflict detection unit

5.3.1 Operation related conflicts

5.3.1.1 Conflicts in the context of a transaction retrieve or change operation:

For every client receiving a copy of a server model, an additional copy of each delivered model is kept at the server side. We could say the state of the all models delivered to a particular client is preserved at the server side in a pool of delivered models. For every client

one pool of delivered models is kept at the server. The following figure illustrates the statements made:

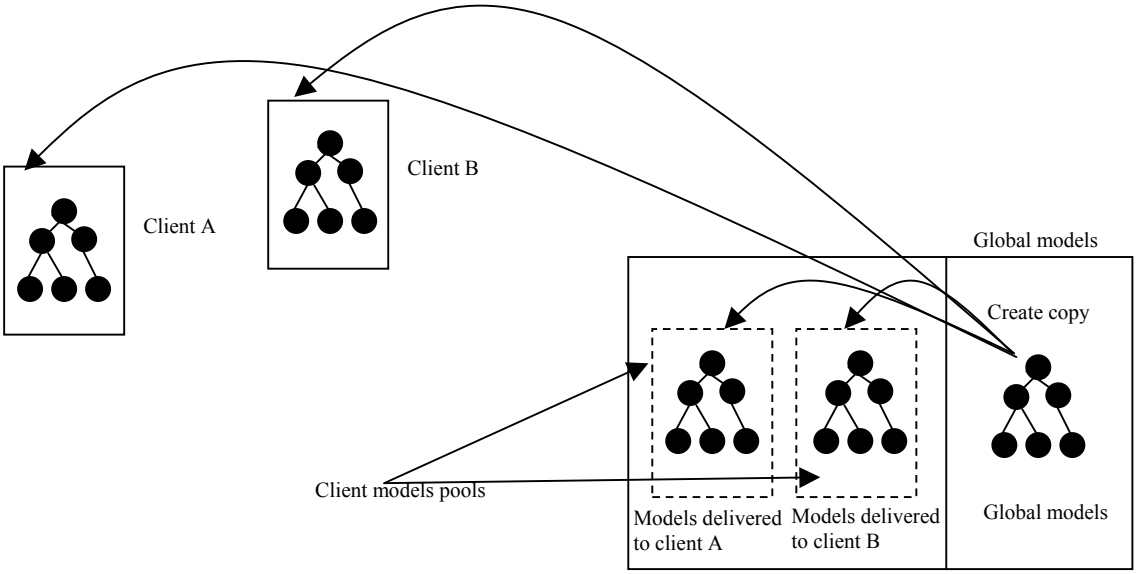


Figure 15: Server side architecture

I will refer from now on to the additional copies of models held for a particular client at the server side as the client's pre-image.

Global models represent shared data items that could be manipulated concurrently by multiple clients. A transaction of a particular client is always replayed at the server side and involves global models and the corresponding models of the client's pre-image. A model retrieve operation of a replayed transaction retrieves a particular global model and the corresponding model of the client's pre-image, including all of its child models. The two retrieved models can be compared to see if a concurrent client has manipulated the global model in the meantime. If the global model differs from the corresponding model of the client's pre-image then the model state delivered to the client does not match the current global model anymore. This implies that a concurrent update by a different client must have happened. As already indicated the comparison of the models takes place after the retrieve operation is performed. If the conflict detection unit compares two models, it is actually performing a comparison of model sub trees. The extracted model of the client's pre-image and the corresponded server model are actually **root nodes from where the model comparison starts.**

By comparing the client's pre-image sub tree with the server's sub tree the following possible conflict types can be detected.

- **A concurrent client changed an existing model:**
  Should the client's pre-image sub tree be part of a comparison containing any models with a different state than the correspondent server models then a concurrent update by a different client must have occurred.

- **A concurrent client added a new sub model:**
  Should the server tree contain any additional models at each level of a tree below the current root node that cannot be associated with a correspondent model of the client's pre-image tree then a concurrent client must have added a new child model to server side.

- **A concurrent client deleted an existing sub model:**
  Should the client's pre-image tree contain any additional models at each level of a tree below the current root node that cannot be associated with a correspondent server model then a concurrent client must have deleted the corresponding model at the server side.

In the Figure 16 below, two concurrent clients are synchronized with a server model named X of state n. Model X is a leaf model and client A and B are locally changing model X_n to X_{n+1}. Client A transfers its transactions log to the server. The transaction of client A is replayed and no conflict is detected. After client A's transaction was processed, client B's transaction is also transferred and replayed at the server. The server system detects a conflict since the global model state was change to X_{n+1} by the first transaction therefore not matching the state of the corresponding model of client B's pre-image:

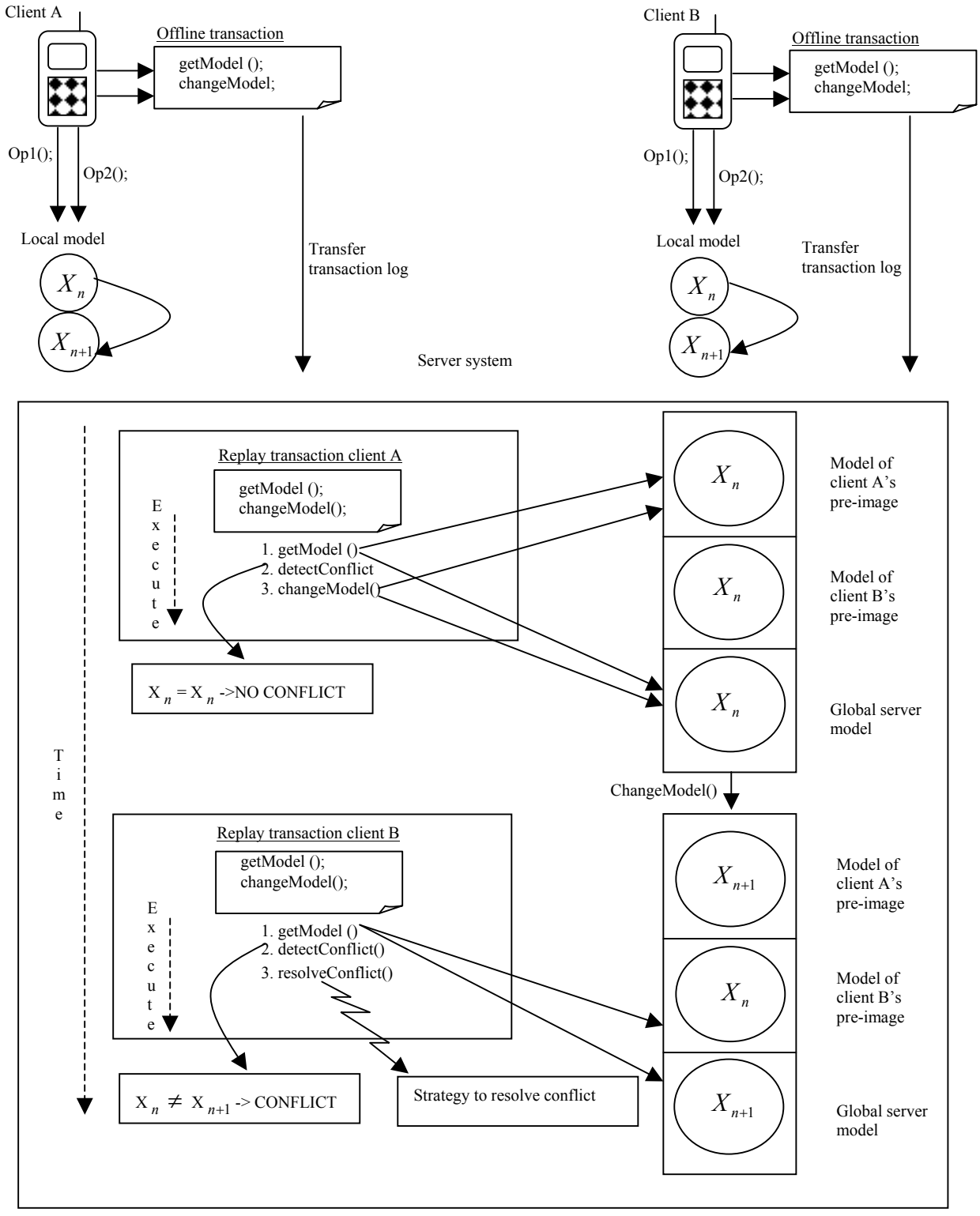


Figure 16: Concurrent update of a unique model by two clients

If possible conflicts are detected, the conflict-resolution unit tries to resolve the conflicts in the context of a specific retrieve operation. It is quite possible that the conflict-resolution unit decides to ignore all conflicts in order to handle them in the context of the next operation of the transaction being executed. Conflicts detected after a retrieve operation could for example

be handled and solved within the context of a possible next change operation. One advantage of this approach is that operation semantics including input parameters or other values could be taken into account to solve any conflicts automatically (see section 2.3.2). Before a possible change operation transfers the models of the client's pre-image and the global server model into a new state, the current server model state $S_{Server\_Model\_n+1}$ and the current state $S_{Client\_Model\_Pre-image\_n}$ of the corresponding model of the client's pre-image are temporarily saved. After the change operation has been concluded, the state $S_{Server\_Model\_n+1}$ is changed to $S_{Server\_Model\_n+2}$ and the state $S_{Client\_Model\_Pre-image\_n}$ is changed to $S_{Client\_Model\_Pre-image\_n+1}$. The prior saved states $S_{Server\_Model\_n+1}$ and $S_{Client\_Model\_Pre-image\_n}$ are compared after the models are changed to redetect all conflicts that occurred during the model-retrieving phase. It is essential to compare the unchanged models after the change operation has been completed to redetect any conflicts since it is possible that the change operation would transfer the model of the client's pre-image and the corresponding global server model to an equal state. Since conflict detection is based on finding any differences between two models no conflicts will be detected if the change operation transfers both models into an equal state.

The conflict detection unit always produces a conflict history vector containing conflict description objects should any conflicts be detected. Model conflict description objects could include a reference to the model(s) concurrently changed added or deleted at the server side, a unique model path describing the type and the ID of a model object. The figure below demonstrates how a conflict history vector is constructed by detecting differences between a global server model and the corresponding model of the client's pre-image:

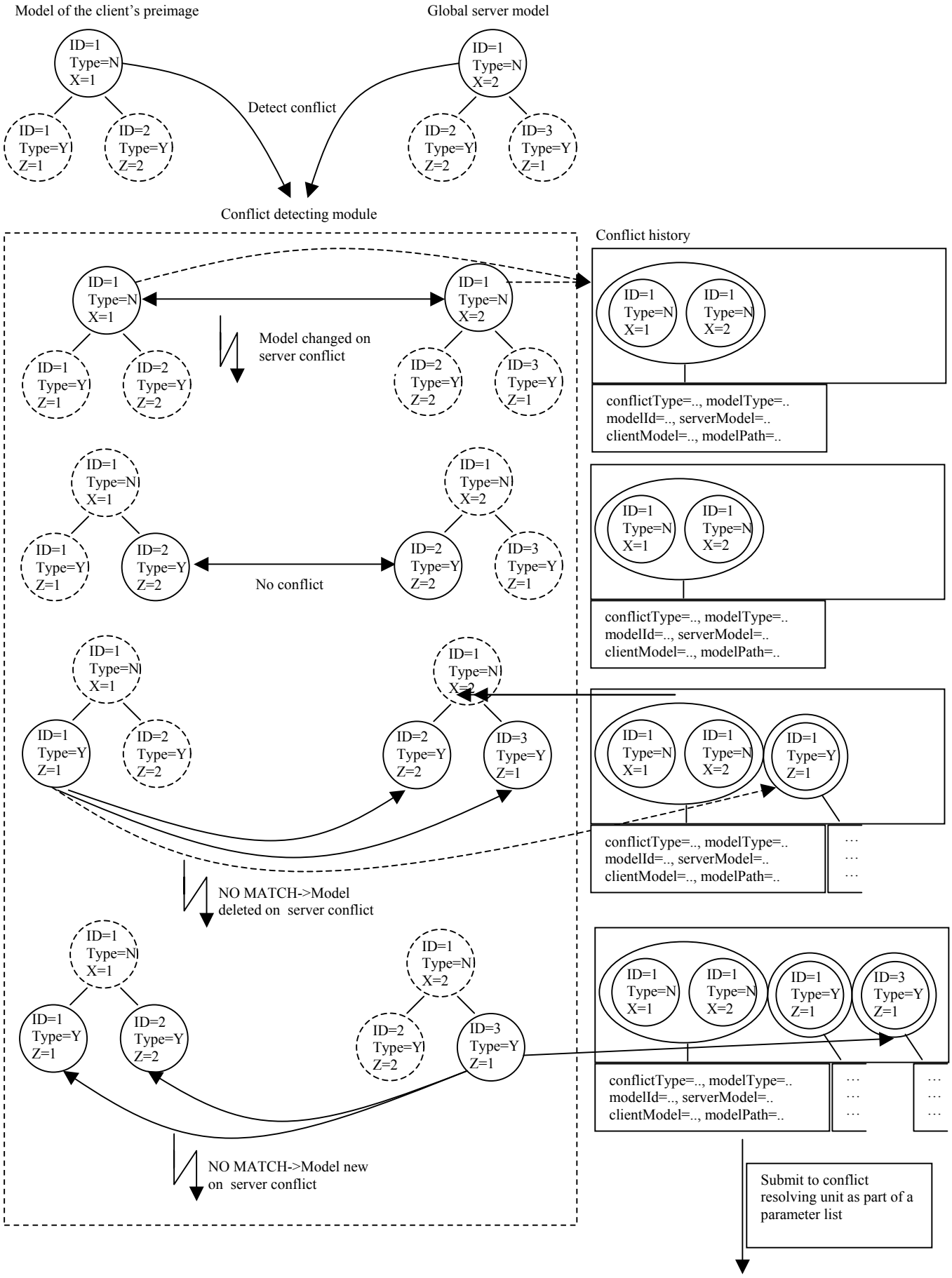


Figure 17: Detecting a conflict

Figure 17 demonstrates how a model of the client's pre-image with ID=1 of type N and the corresponding server model are inspected for any conflicts. The conflict detector recognizes that the attribute X contains different values and therefore a concurrent client must have changed the global server model. A model compare result object, containing conflict related data such as the type of conflict, is created and inserted into a conflict history vector. The comparison process also includes all child models. As indicated in Figure 17 the client's pre-image child model with ID=2 of type Y and the associated global server model contain equal attribute values and therefore no conflict is detected. No corresponding server model can be found for the client's pre-image child model with the ID=1 of type Y. This indicates that a concurrent client must have deleted the corresponding child model at the server side. As result of detecting a deleted server model, a compare result object is inserted into the conflict history vector. The inserted object contains the type of conflict, a reference of the deleted object (reference to the client's pre-image model) and other conflict related data. An additional conflict can be detected because no corresponding model of the client's pre-image can be discovered for a server model of type Y with a model ID of 3. The only conclusion is that a concurrent client must have added this new model to the server. If any conflict can be detected, the conflict-resolving unit is activated with the conflict history and operation related values to resolve the conflict. As the example demonstrates a difference in a client's pre-image model and the corresponding server model could produce multiple entries in the conflict history vector.

The conflict-resolving unit will decide how to react to conflicts in the context of a specific operation. It should be noted that a model fetch operation always precedes a model change operation therefore a conflict can only be handled in the context of a change operation if the conflict resolving unit decides in the preceding retrieve operation to ignore the different model states fetched. The following scenario backs up the statements made:

Two concurrent clients A and B are changing the remarks of the same report model while being offline. Local transactions are recorded and the transaction log file of client A is transferred to the server and replayed without a conflict because the involved model of the client's pre-image still matches the corresponding global model of the server after the model retrieve operation is completed. Now the transaction log of client B is transferred to the server and a retrieve operation at the beginning of a transaction retrieves the client's pre-image model and the corresponding server model. The retrieved copy represents the state of the

mobile device's version of model at the time of delivery by the server. The same retrieve operation is also applied to the data storage unit containing the corresponding global server model. A conflict is detected since the two models differ therefore a concurrent update by a different client must have taken place. The conflict detection is configured to ignore the conflict at this point because the conflict should be handled in the context of the following change operation. The framework saves the current states of the involved models ($S_{Server\_Model\_n+1}$ and $S_{Client\_Model\_Pre-image\_n}$) and executes the next change operation of the transaction. The conflict detection unit is comparing the model states $S_{Server\_Model\_n+1}$ and $S_{Client\_Model\_Pre-image\_n}$ after the change operation. A conflict is redetected if we assume that both clients stored different remarks in the models. The conflict-resolving module decides that the remarks of both transactions should be concatenated, since the conflict handler is provided with all necessary data.

5.3.1.2 Conflicts in the context of a transaction create operation:

In the current version of the application framework a conflict as a result of creating two completely identical objects is not possible since the data access unit always provides a unique model-ID for every created object. Therefore the state of two objects can never be completely identical.

5.3.1.3 Conflicts in the context of a transaction delete operation:

If a replayed transaction involves the deletion of a model at the server side the application framework retrieves the model to be deleted and checks if a different client concurrently changed the model before the deletion process continues. In case the model out of the client's pre-image is outdated or the model does not exist anymore at the server side the conflict-resolving unit resolves this conflict.

5.3.2 Model-only related conflicts

Conflict detection covered so far is restricted to models being part of an operation. The conflict detection unit would be unable to detect conflicts caused by concurrently changed global server models if the changed server models are not used by operations of the current transaction. The conflict detection and resolving-unit would remain inactive until the affected

models are referenced by operations of a client's transaction being replayed at the server side. The conflict detection should be extended to detect this class of conflicts at the next transaction replay phase. This can easily be accomplished by comparing the complete global server tree with the complete client's pre-image tree after the transaction is replayed and operation related conflicts are thus detected and resolved. The conflict detection unit would create a conflict history vector containing description objects of conflicts not detected and not yet resolved. I am referring to this set of conflicts as model-only related conflicts. If desired, the detection of model-only related conflicts can be switched off in a configuration file.

5.3.3 Alternative approach

Some distributed offline systems use a version number schema to detect possible conflicts. The server could for example memorize a vector of version numbers, representing versions of data items delivered to a particular client. The client receives the pre-image version vector together with all data items from the server. The mobile device could change some data items and transmits the pre-image version vector together with all changed data items including a new version vector back to the server. The new version vector might contain incremented version number of data items changed. The server can detect a concurrent update conflict by comparing the pre-image version vector received from a client with its current version vector. If the version number matches, no concurrent update took place and therefore all changed data items and the new version vector can be saved persistently. Many variations of this approach can be found in distributed off line system. One of the main disadvantages of this approach is that all semantics of any operations performed at local devices are lost and cannot be used in a conflict-resolving schema. However a combination of replaying transactions and using version vectors could also be used in order to include operation semantics in a conflict-resolving schema. The server could memorize a version vector V_{ci} where c,i $\in$ {1..n} for every model delivered to a client and a version vector V_s where s $\in$ {1..n} for all global server models. The version of every data item of a model delivered to a client would be saved at V_{ci}[index item]. The current version of data items of the server models would be stored at V_s[index item]. The client could record a local transaction that is replayed at the server side on the involved global models and while being in replay mode the server could verify that every operation performed on a data item of a server model would result in an increment of a version number in the correspondent server and client vector. Concurrent updates could be

detected by discovering unequal sequence numbers in the client vector and correspondent server vector.

5.3.4 Summary

Conflicts can be defined as differences between the state of a global server model and the state of a correspondent model of the client's pre-image. A conflict type describes the kind of difference detected when comparing model states. The comparison of models always includes the comparison of all child models. Conflicts discovered while executing operations are called "operation related conflicts". Differences can also be detected by comparing the absolute root node of the client's pre-image tree with the absolute root node of the server's model tree. Since all child nodes are also compared all differences between the two trees can be detected and only the conflicts not recorded while replaying a transaction are memorized. Conflicts detected as a result of comparing complete server and client pre-image trees are called "model-only related conflicts". The figures below outlines the differences in discovering an operation related conflict and a model-only related conflict:

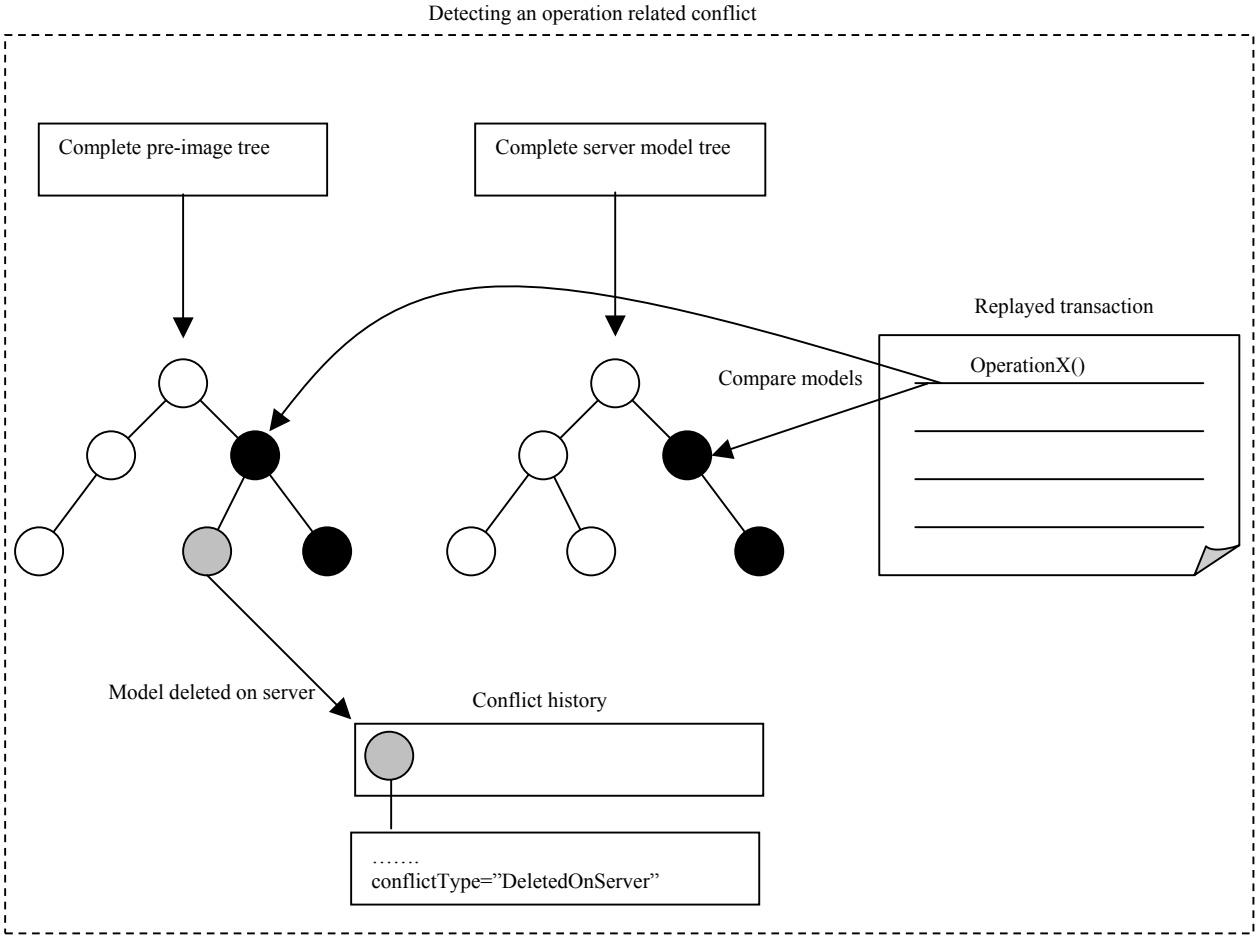


Figure 18: Detecting an operation related conflict

In Figure 18, a server model and a model of the client's pre-image are compared. The two models are referenced by an operation of a replayed transaction. As illustrated the two models are nodes of an overall tree structure. All child models of the nodes being compared are included in the comparison process. A conflict of the type "DeletedOnServer" is detected since the model of the client pre-image contains a child for which no counterpart exists at the server system. This implies that another concurrent client deleted earlier the missing node on the server system. A conflict description object is created and inserted into a conflict history vector that is submitted to the conflict resolving module

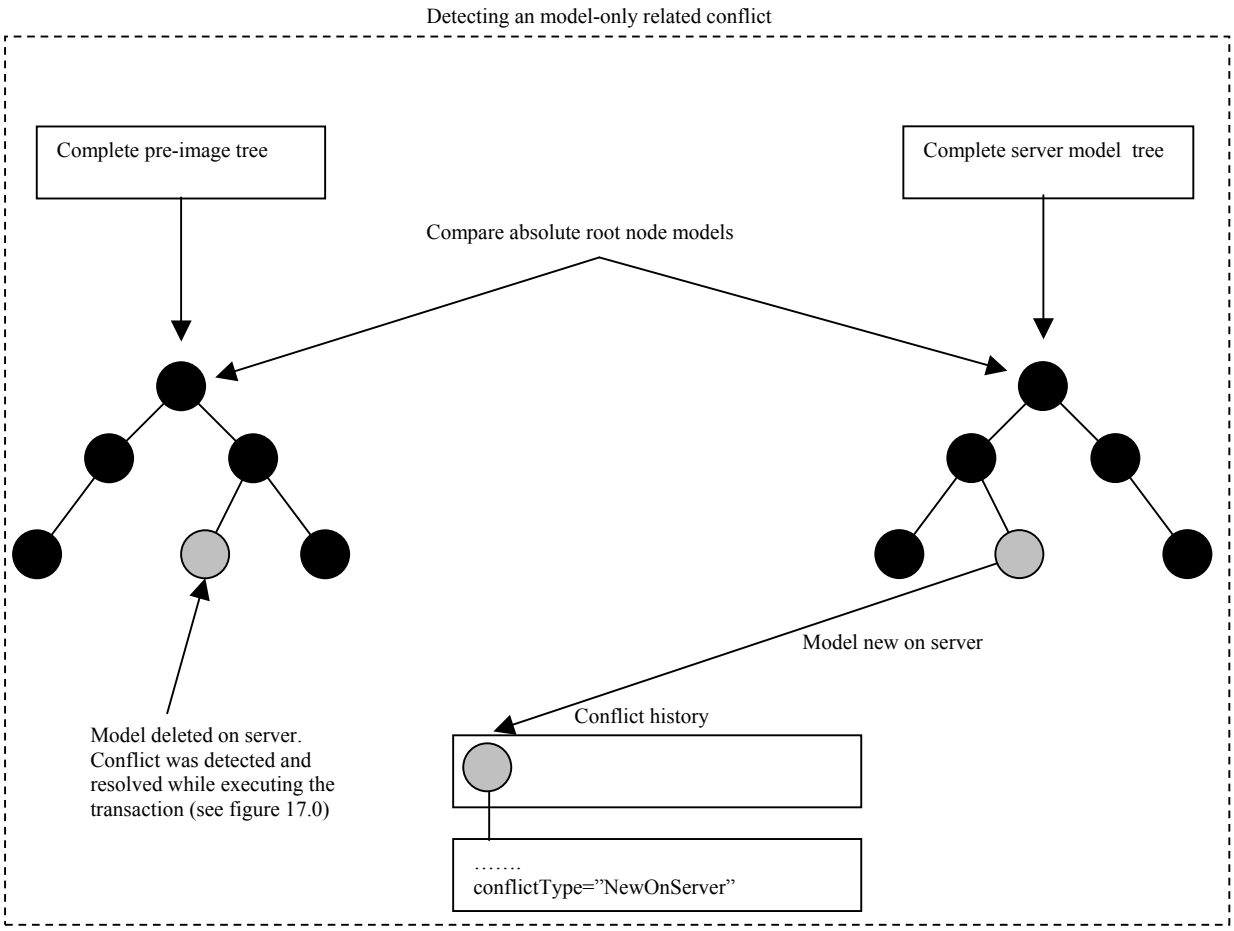


Figure 19: Detecting a model-only related conflict

Figure 19 illustrates how a complete server tree and the client's pre-image tree are compared to discover conflicts not being detected by the conflict detection-unit when an earlier transaction was replayed. Model-only related conflicts are memorized in the conflict history vector and already processed operation related conflicts are ignored.

5.4 Design of the conflict resolution unit

The main requirement of the conflict resolving unit is that conflict resolving must be flexible and fully configurable by an XML-configuration file. I decided to provide the mobile application developer with a clean interface that permits the developer to plug in his/her own conflict handler routine for a particular conflict. The developer can describe a particular conflict and assign his/her own conflict handling routine to it by adapting an XML-configuration file. A mobile application developer can specify two types of conflict handlers:

- Handlers reacting to operation related conflicts

- Handlers reacting to model-only related conflicts

The strategy used in resolving conflicts depends on the conflict handler's code. The application framework reads the XML-configuration file while being in an initialization phase and builds a hash table for operation related conflicts and a hash table for model-only related conflicts. In each hash table conflict description objects are assigned to particular conflict handlers. If operation related conflicts are detected the conflict-resolving unit is activated with operation related data, a client-ID identifying a mobile device and a conflict history vector. Only the client-ID and the conflict vector are submitted to the conflict resolving-unit in case model-only related conflicts are discovered.

Operation related data include:

- The name of the operation and all input parameters
- The state of the global model after the operation is executed
- The state of the corresponding model of the client's pre-image after the operation is applied
- A client ID identifying a mobile device

The instance data of a conflict description object include:

- The type of the conflict that occurred
- The involved server model and the associated model path
- The involved model of the client's pre-image and the associated model path

- The involved model type
- The involved model-ID

Entries in the conflict history vector represent a complete description of a particular conflict detected. If conflicts are not ignored, the conflict-resolving unit extracts all conflict description objects and searches for every single object the corresponding conflict handler in the hash tables. Every extracted conflict handler is executed to resolve the conflict. The following figure illustrates the basic architecture:

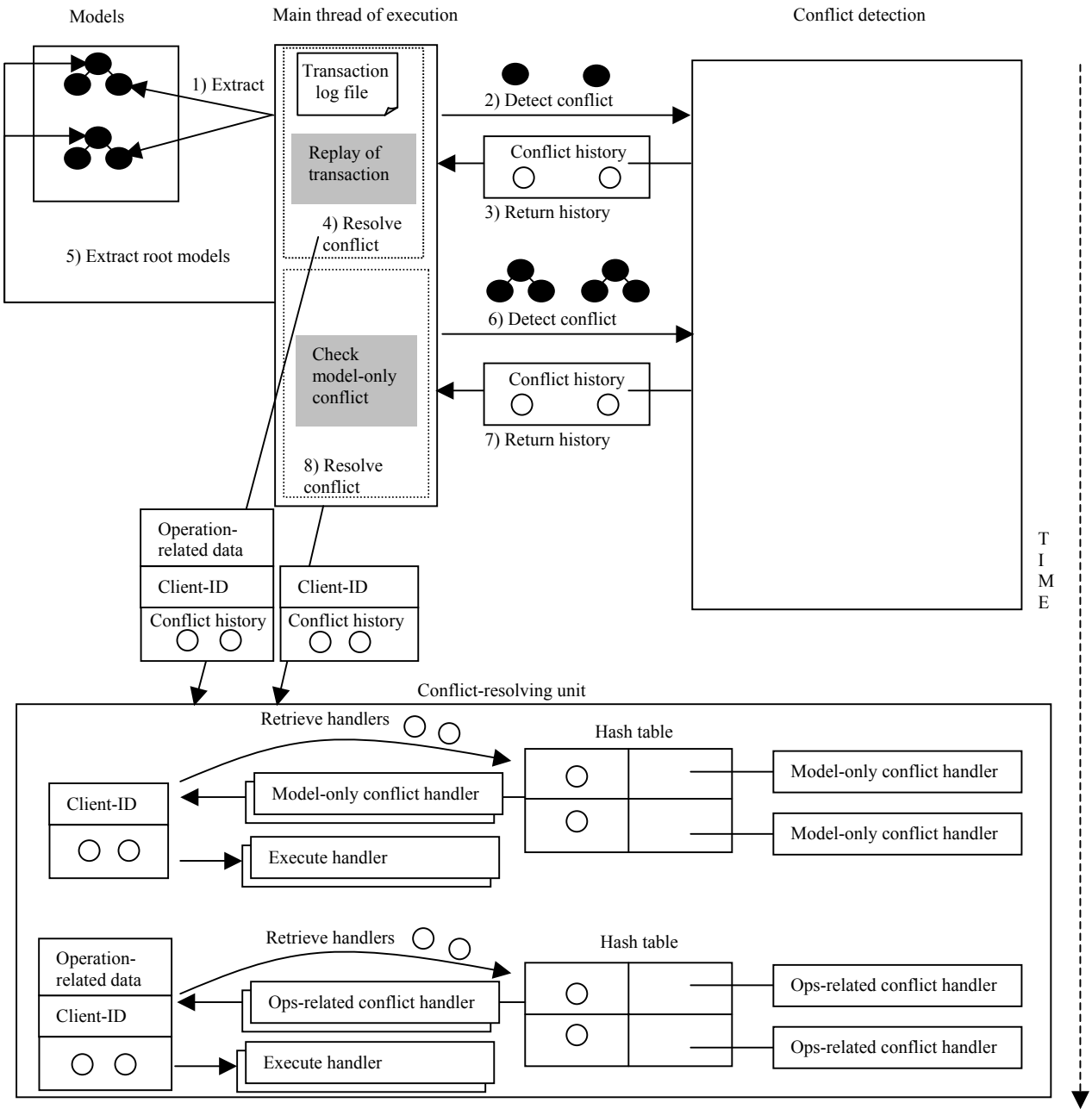


Figure 20: System architecture

Figure 20 presents the overall system architecture. The main thread of execution contains two code sections. The first code section replays a transaction, detects operation related conflicts and resolves them. The second code section is responsible for detecting and resolving model-only related conflicts. A number denotes the order of the activities performed. The conflict-resolving unit is activated in case a conflict is discovered and all relevant parameters are submitted to it. As displayed in figure 20, the conflict-resolving unit extracts the conflict handlers in compliance with the parameters submitted. The handlers are extracted from the appropriate hash tables by comparing the keys of the hash table with the submitted conflict description objects. Once the handlers are extracted, they are executed and the conflict description object and other parameters are submitted.

5.4.1 Alternative Approach

Another approach that could have been chosen is that the conflict resolving unit provides a fixed set of rules to the application developer that describe a specific strategy to resolve a conflict. A configuration file could contain entries that define what strategy should be used if a server is concurrently updated. The strategy could be “server model wins” or “mobile device model wins”. Providing rules in the form of pre or post conditions could also include the semantic of an operation. However this approach has the major disadvantage that a set of predefined rules adapted or included in a configuration file is usually incomplete and not suitable for every possible application scenario. The specification of a new language would be necessary that defines the syntax of the conflict resolving rules including a possible composition of different rules to provide the flexibility required. Why defining a new language to define strategies that are used in case conflicts are detected? I decided that the best solution would be to let the developer specify conflict-resolving strategies by providing rules in form of Java code, a language powerful enough to cope with every possible scenario. A conflict handlers code could be seen as a set of rules written in Java that specify how conflicts should be resolved. Different handlers can be written and plugged in by the developer. This approach is not really new. As outlined in chapter 2.2.3 Palm pocket computers are using conduits that specify how a possible conflict should be resolved. Conduits can by written in C or C++ and are plugged into the application that might cause conflicts. Conflict resolving strategies are specified in form C or C++ code.

5.4.2 Summary

The conflict resolving approach used in this Master's dissertation could be regarded as event triggered. The conflict detection unit generates events in form of conflicts detected and the conflict resolving unit provides the infrastructure to handle them by redirecting some of the input parameters received from the outside to an appropriate conflict handler. The conflicts that the mobile application developer is interested in detecting and resolving can be specified in an XML-configuration file.

5.5 Implementation of the conflict detection unit

5.5.1 Client Pre-Images

The detection of conflicts requires the existences of client pre-images on the server system. For every mobile device that requests global models from the server for the first time, a subdirectory is created on the server's file system. The states of models delivered to a client are saved in an XML-file in the earlier created subdirectory. Model objects cached in memory represent the client's pre-image used during runtime and are periodically preserved in the XML-file. On shutdown and restart of the server system, cached objects can be reconstructed from the saved file. A mobile device always receives models in form of an XML-file. After a mobile unit receives a file, it is parsed and model objects are created in the main memory. Delivering models in an XML-format instead of a binary format has the advantage that no complex marshalling and de-marshalling of data is necessary. Mobile clients require fewer resources to parse a received XML-model file than to de-marshal received binary data. On the other hand transmitting model data embedded in XML-files requires more bandwidth than transmitting them in a binary format.

Figure 15 from chapter 5.3.1.1 can be extended to:

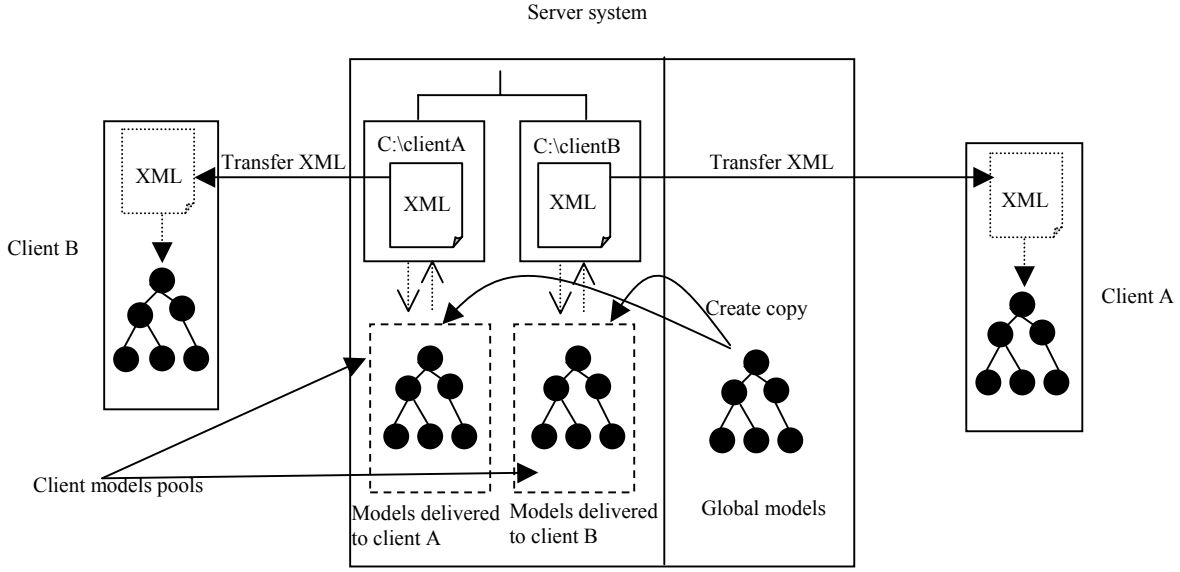


Figure 21: Client pre-image models

5.5.2 Visiting and comparing models

Comparing a global server model and the corresponding model of the client's pre-image also include comparing all of their children. An algorithm is needed to traverse nodes of both trees synchronously. I am using a modified version of the "level order traversal" algorithm as described in [9]. A similar code fragment as shown below is used in [9] to explain the level order traversal algorithm:

```
void traverseTree (Node node)
{
        NodeQueue queue = new NodeQueue();
        queue.put(node);
        while (!queue.empty() )
        {
                node = queue.get();
                node.item.visit();
                if (node.left !=null)
                {
                        queue.put (node.left);
                }
                if (node.right != null)
                {
                        queue.put(node.right);
                }
        }
}
```

Figure 22: Level order traversal code

The code fragment in Figure 22 uses a queue to store nodes that should be visited. A queue represents a FIFO (first in first out) abstract data type. As long as the queue contains nodes not yet visited, a node is extracted from the front of the queue and visited. The extraction of a node implicitly removes the node from the front of the queue. If the extracted node contains a reference to a left or a right child node then the child nodes will be stored at the end of the queue for the next iteration. The queue is filled with the root node of the tree before the iterations start. The tree traversal can be illustrated as followed:

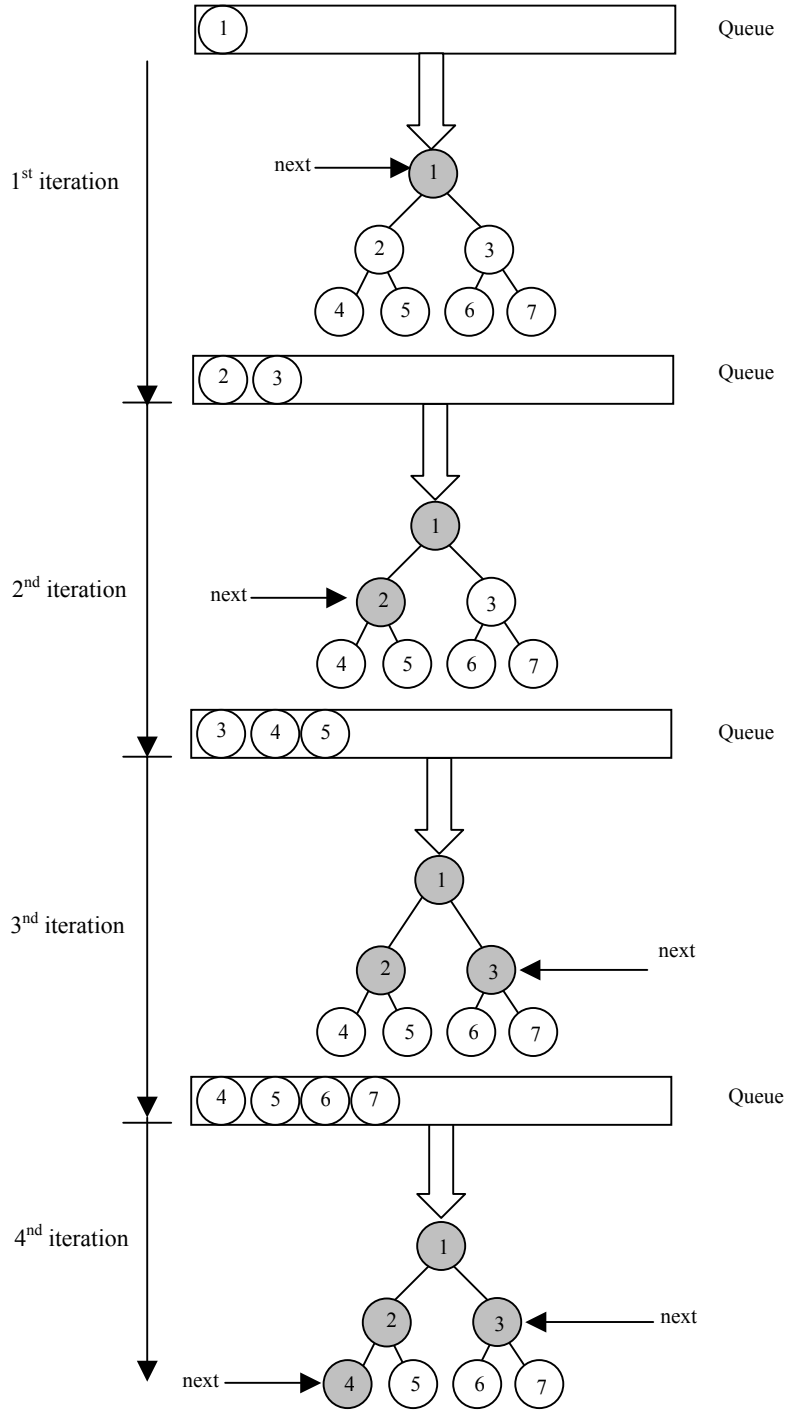


Figure 23: Graphics of level order traversal algorithm

Some minor modifications of the “level order traversal” algorithm are necessary so that a global server model and the corresponding model of the client’s pre-image can be submitted and compared. The algorithm compares two models that contain the same model-ID on each level of the two trees. If the comparison process detects that some of the attributes of two compared models are different, a conflict description object is generated and memorized. If the algorithm detects a model on a tree level that cannot be associated with a model on the corresponding other tree then a concurrent client must have deleted or created a model on the server. As a result a conflict description object is generated and memorized. The algorithm can be formulated in pseudo code as followed:

```
void traverseAndCompareTrees (server model, client model)
{
    Add the server model to a “server model queue”
    Add the client model to a  “client model queue”

    while (“server model queue” and  “client model queue” are both not empty)
    {
        Retrieve and remove server model from “server model queue”
        Retrieve and remove client model from “client model queue”
        If (the server model node has a different state than the client model)
        {
            Generate a conflict description object and memorize it. -> Concurrent change of models by other client
        }
        Retrieve all child models from server model and store them in a “server children vector”
        Retrieve all child models from client model and store them in a  “client children vector”
        while (A pair of models with the same ID can be retrieved from “server children vector” and “client children vector”)
        {
            Save found server child of found pair in “server model queue”
            Save found client child of found pair in  “client model queue”
            Remove server child of found pair from “server children vector”
            Remove client child of found pair from “client children vector”
        }
        if (“server children vector” contains any models)
        {
            Generate conflict description object and memorize it ->Concurrent client added a server model(s)
        }
        if (“client children vector” contains models )
        {
            Generate conflict description objects and memorize if -> Concurrent client deleted server model(s)
        }
        Clear all vectors.
    } // end while
}
```

Figure 24: Modified level order traversal algorithm to visit nodes and detect conflicts

The pseudo code in Figure 24 does not differ significantly from the “level order traversal“ algorithm. The algorithm uses two queues to store models of an associated model pair that should be next visited and compared. Each queue contains one model of a model pair. After extracting and removing the models from the appropriate queues, the attributes of the server model and the attributes of the model of the client’s pre-image are compared and checked for a conflict. Since the two models could have more than two child nodes, all children are fetched from the two nodes and stored in two different vectors. The algorithm tries to find in the two vectors as many matching pairs of models with the same model-ID as possible and stores each model of a found pair in the appropriate queue. All found pairs are removed from the two vectors to detect any models for which no associated counterpart exists on a particular tree level. Conflicts are detected if the vector that stores children of the server model contains any children after all matching pairs are removed. In that case no associated child model of the client’s pre-image can be discovered at the current tree level and therefore a concurrent client must have added a new server model. Should the vector storing children of the client’s pre-image model contain any models after all model pairs are removed, then the current level of the client’s pre-image tree must contain models for which no counterpart exist at the corresponding level of the server model tree. This would imply that a concurrent client must have deleted a model at the server tree.
In case a conflict can be found, a model path and other conflict related data is stored in a conflict description object. The model path can be used in the resolving unit to describe a path from the root node to the model node involved in a particular conflict.

It is important to note that the algorithm presented depends on the fact that only models of the same type are stored on a level of a tree (see Figure 8.0).

The following graphical representation tries to explain how the algorithm works. The numbers within the circles represent the model-ID:

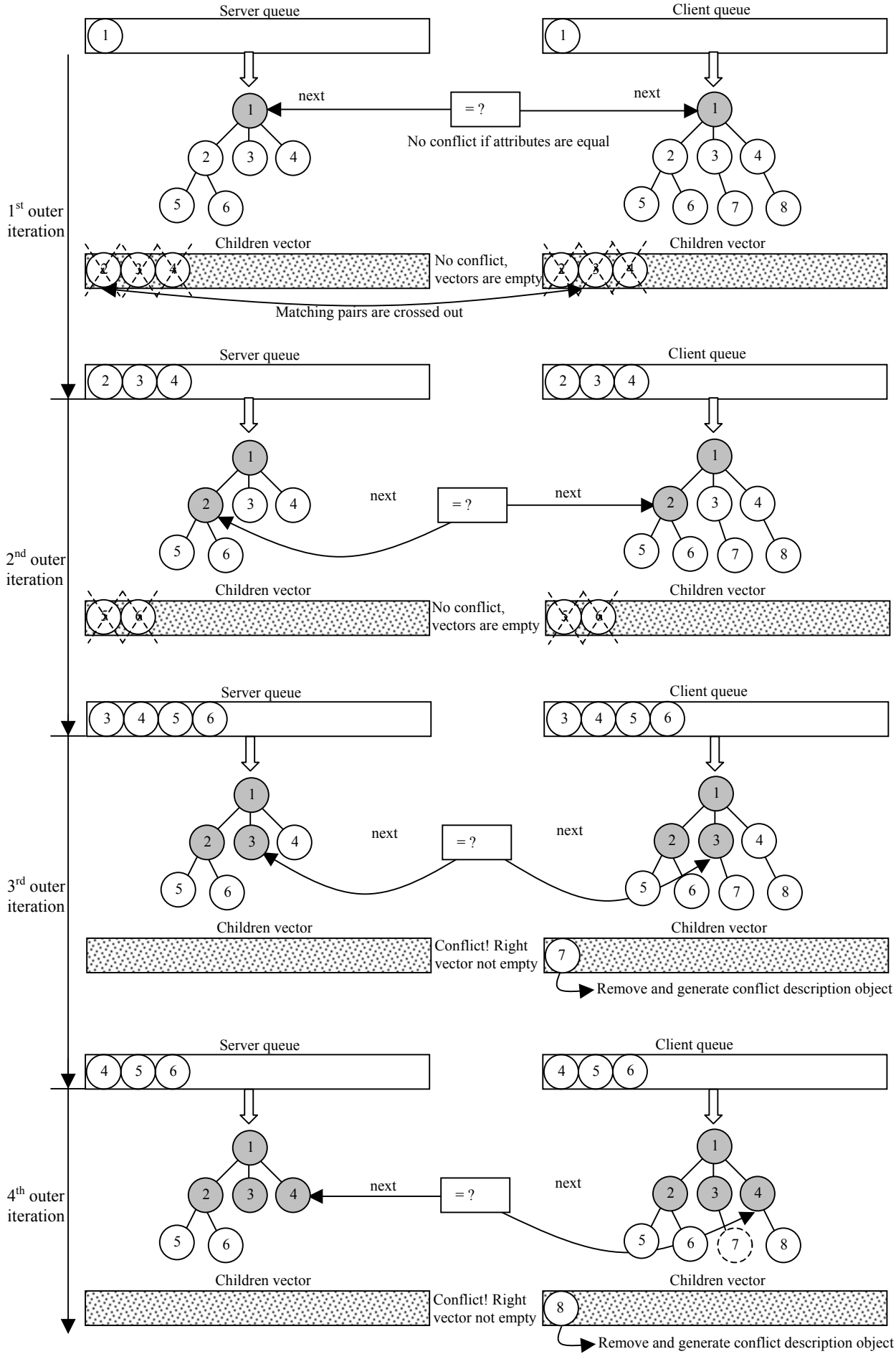


Figure 25: Graphical representation of modified level order traversal algorithm

5.6 Implementation of the conflict resolution unit

As already mentioned the conflict resolving unit reads an XML-configuration file during the initialization of the framework (see chapter 6.4). The file contains descriptions of possible conflicts and associated conflict handlers. The XML-configuration file contains two main XML-tags:

- Tags to describe operation related conflicts
- Tags to describe model-only conflicts.

The application developer must provide one of the mentioned XML-tags for every conflict that should be intercepted. Tags for operation related or model-only related conflicts contain a child tag with attributes and corresponding values that represent a complete description of the conflict that the developer wants to be resolved. The child tag itself contains another child tag that specifies the name of a Java class used as a conflict handler. An XML-parser reads the configuration file and builds a DOM (Document Object Model)-tree in the main memory. A DOM–tree is a tree structure of the XML-file in memory. The conflict-resolving unit uses the DOM-API to retrieve elements, attributes and attribute values of the DOM-tree in order to build a hash table consisting of key value pairs. The keys of the hash table are conflict description objects with attributes and values matching the attributes and the values of the XML-tags. The values of the hash table are instances of conflict handler Java classes. The Java reflection API is used to load a handler class into the memory and to create handler instances during the build-up of the hash table. Every handler class must implement a predefined Java interface to guarantee that it provides a conflict resolution method that is required by the conflict-resolving unit. Only then can the handler be retrieved from the hash table and be plugged into the conflict-resolving unit during runtime.
The implementation of the conflict-resolving unit extracts conflict description objects of the conflict history vector submitted by the conflict-resolving unit and tries to match all attributes of the a conflict description object with attributes of the description object used as a key in the hash table. If a match is found, the handler is extracted from the hash table, plugged into the conflict-resolving unit and executed.

The following figure shows operation and model-only related conflict tags, used to describe a particular conflict interested in:

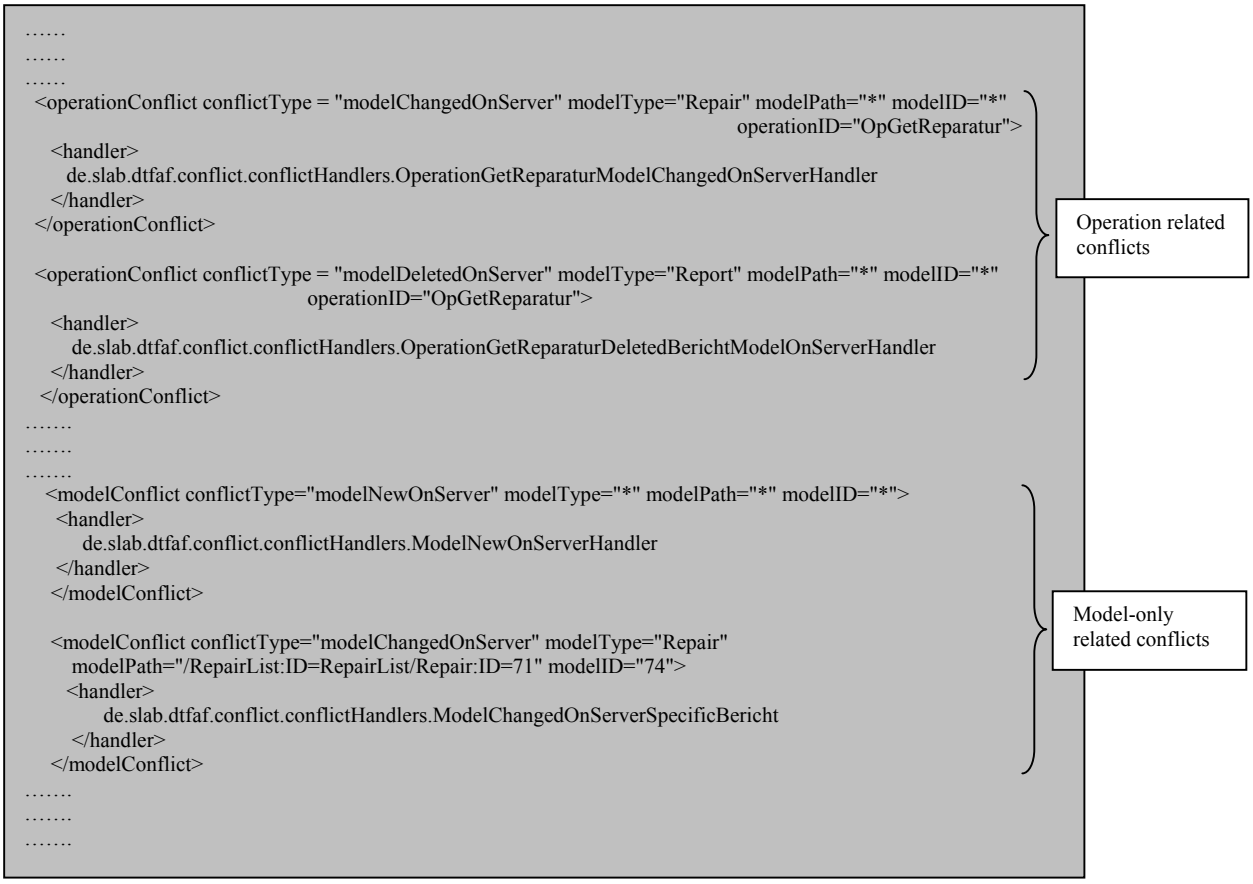


Figure 26: XML-configuration file to describe conflicts

Figure 26 shows different attributes used to describe a conflict. For the handler to be executed, the attributes of the XML-file must all be equal to the attributes of the conflict description object submitted by the conflict detection unit. A wildcard * indicates that the attribute is ignored while comparing the conflict description objects.

The attribute used for operation and model-only related conflicts are:

- conflictType:
  This attribute describes the type of the conflict the developer is interested in intercepting.
- modelType:
  Conflicts are intercepted if the type of the model object involved equals the type named by this attribute.

- modelPath:
  Conflict are intercepted if the location of model objects involved equals the path specified by this attribute
- model ID:
  Conflicts are intercepted if the model-ID of the model object involved equals the model-ID declared by this attribute.

The operation-ID attribute is only relevant for operation related conflicts and specifies that a developer is interested in conflicts of a particular operation.

6. Testing the prototype in combination with a back-office system and Teleservice

The Teleservice application analyzed earlier is used to test the conflict detection and resolving unit. The dtF/SQL relational database system, developed by the sLAB corporation, is used as a back office system. A back office system could be any application for which a Java data access layers can be built. As already mentioned in earlier chapters a data access layer represents an interface to the outside world. The data access layer provides methods that are called by the framework to manipulate models stored in an external system. Since a relational data base system is used as a back office system, the data access layer has to map method calls that retrieve, change delete or create models to calls of the database SQL-API. If methods of the data access layer are activated, a connection to the data base system is established and SQL-commands are transmitted to the database system. The data access unit acts as a converter that converts method calls and model parameters to SQL-calls that are submitted to the data base system. The database access is accomplished by using the Java-JDBC API that guarantees a platform and database independent access API. In order to test the conflict detection and conflict-resolving unit a second data access layer is needed to retrieve or manipulate any models from the client's pre-image. If transactions are replayed the framework synchronously activates the back-office data access layer and the access layer of the client's pre-image to manipulate global models and models of the client's pre-image.

The scenario could be illustrated as followed:

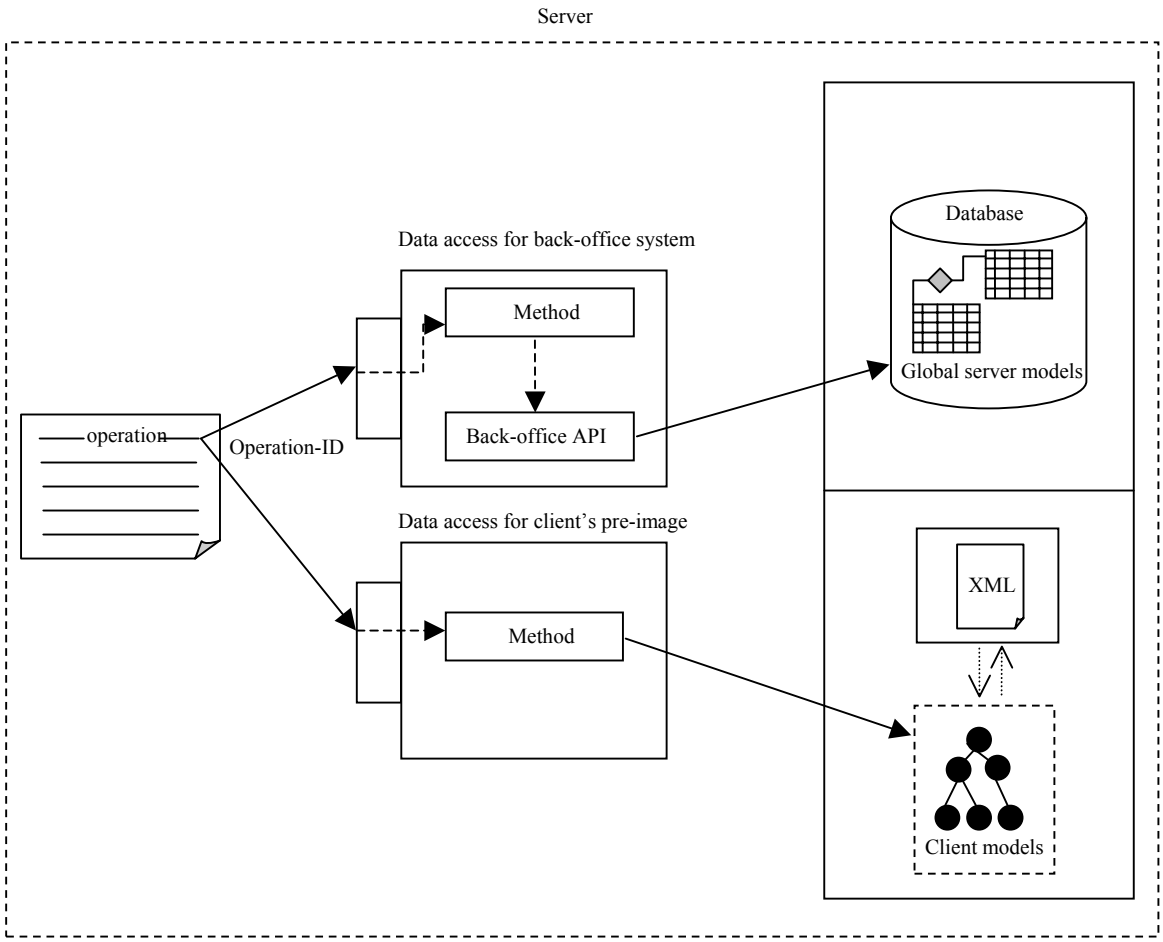


Figure 27: Data access layers

Figure 27 shows how an operation that is replayed at the server triggers, the execution of the corresponding methods in two data access layers. As indicated in Figure 27 all global models are held in a relational database system. The data access layer for the Back-office system is using the database access API to retrieve, change, delete or create global server models.

Most back-office systems provide a C-API that could be used by the Java's JNI (Java Native Interface). A data access layer could, for example, manipulate models by providing Java methods that call appropriate C-native functions of a back-office when being activated.

The conflict detection and resolving unit works well in combination with the Teleservice application. Mobile clients, running the Teleservice application and containing over a 100 models are used to verify the correctness of the conflict detection and resolving units. Automated test cases are also written to test the conflict detection and resolving unit isolated from the rest of the system. Having automated test cases that feed the conflict detection unit

with predefined server models and models of the client's pre-image, the possibility of the behavior of the implementation being inconsistent with the requirements is reduced. A framework called JUnit is used to create and run the different automated test cases.

7. Conclusions and recommendations for further work

7.1 Summary

This Master's Dissertation outlined how global data inconsistency that resulted from concurrent updates by multiple clients can be detected by comparing client and global object states. The mobile application framework and the Teleservice application provided the infrastructure to develop and test a conflict detection and resolution module. The conflict-resolution unit's approach to intercepting specific conflicts according to definitions made in an XML-configuration file and to resolving conflicts by executing assigned conflict handlers guarantees maximum flexibility in resolving data inconsistencies in different kinds of mobile applications. Operations of a mobile application transaction are preserved by mobile devices in the form of a transaction log that is submitted to the server. A conflict handler running at the server side could use operation semantics, as discussed earlier, to introduce a more intelligent conflict resolution strategy than most distributed off line systems provide.

7.2 Managing the project

The data collected during the interim report provided an overview of the technologies already in place. Some of the approaches reviewed, produced some ideas that influenced the approaches selected in this Master's dissertation. The conduit based conflict resolution of Palm PDA systems that was reviewed during the interim report provided the basis for a similar approach to resolving conflicts in the mobile application framework. The conflict detection unit's design and implementation could be seen as variation of the different time stamp algorithms reviewed in the interim report with difference that the state of data transmitted to the mobile client is preserved in model objects instead of a vector containing numeric values. After a literature survey was completed I made the assumption that estimating the time needed to accomplish the different objectives should be a more or less accurate process. However pride always comes before a fall. As the work evolved it became clear to me that I totally underestimated the time needed to analyze the framework that was built over a 2-year period by multiple software developers. To improve my understanding of the design and the code, I had to reverse engineer some code sections by creating sequence diagrams that improved my understanding of how and why different objects interact with each other. By reviewing the Teleservice application at a later stage, the correctness of the framework was constantly verified and sometimes revised. Therefore more time was spent than forecast investigating the application scenario already in place. The design and the

implementation demanded less time than originally estimated because the literature survey conducted in the interim report not only provided me with ideas of how a possible solution might look like but also helped me to decide which solutions considered in earlier stages might be ineffective in a mobile environment or too costly to develop in the given time frame. Writing up this Master's dissertation was a challenging task. Having to explain why something was done in the way it was done and putting everything into an overall context resulted in a final code review and additional quality assurance tests that were not accounted for when the time plan was constructed.

The following Gantt chart shows the actual time needed and the time predicted to accomplish the objectives outlined earlier:

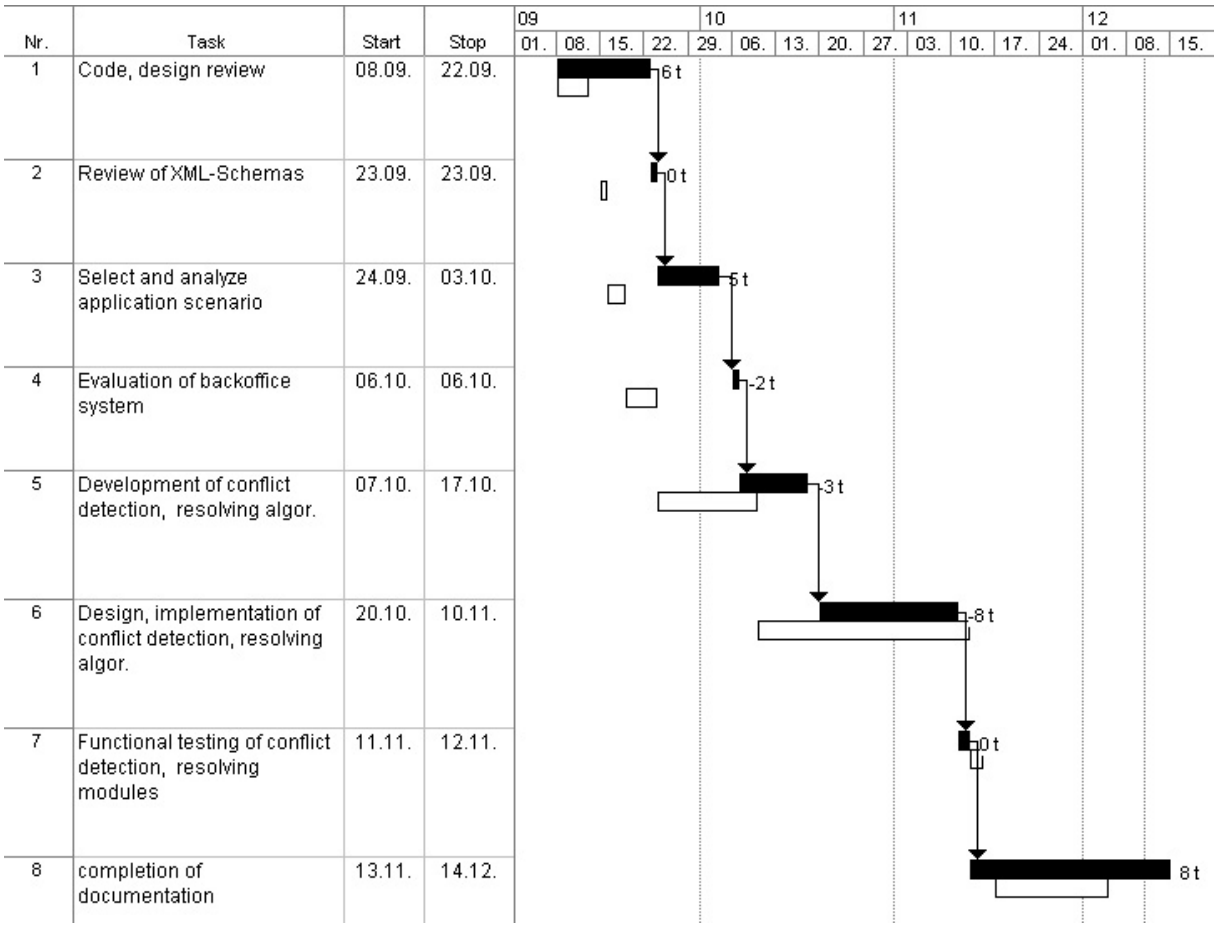


Figure 28: Final Gantt chart

Figure 28 contains white and black rectangles. The width of the white rectangle illustrates the time predicted to accomplish an objective. The black rectangle's width represents the time

that was actually spent on the objective. A positive or negative number denotes the difference in days of the time needed and the time predicted for an objective.

7.3 Conclusions

The approach selected to detect and resolve conflicts proved to be reliable and efficient while testing the Master's dissertation findings within the Teleservice application. However designing and implementing the conflict detection unit was a difficult task since one major constraint was that any found solutions must be fully integrated into the framework. Since the framework was developed earlier, I had to consider details of the frameworks design and implementation while developing a solution in order to avoid the redesign or reimplementation of large parts of the mobile application framework. It was necessary to spend a considerable amount of time analyzing the framework and the Teleservice application in order to integrate the Master's dissertation findings and to prove that the conflict detection and resolution actually works.

Keeping a copy of delivered client models at the server side seemed to be a workable solution with respect to the model driven approach that was selected in the framework's design and implementation. However since every delivered model is cached in the server's main memory, the main memory might be exhausted if a large number of clients are serviced that request an extensive number of large models from the server system. The modified level order algorithm used to detect conflicts proved that conflict detection could be reduced to a comparison of model trees or model sub trees.

Using XML-related technologies to specify which conflict should be intercepted and which handler should be executed by the conflict resolution unit proved to be a very flexible and powerful approach. Verifying the correctness of user created XML-configuration files with the help of XML-schemas and parsing the configuration file with an XML-parser was a straight forward task and was implemented within a very short period of time. The structure of used XML-configuration file can be easily altered in future extensions of the conflict-resolution unit.

The Master's dissertation's findings also confirmed, by using the dtf/SQL relational database as a back office system in combination with the Teleservice application, that the extended framework could be used with any back office system to store global models and to detect and

resolve possible inconsistencies as long as the back office system provides an API that can be addressed by the Java runtime environment.

Several functional tests in combination with the Teleservice application were conducted to verify that the solutions presented are correct and work efficiently however large scale and long term tests involving hundreds of mobile clients will only determine if the presented solution satisfies functional and non functional requirements.

7.4 Recommendations

Replayed transaction log files are processed in FIFO order. To increase the throughput of processed transactions, the concurrent processing of transaction logs should be introduced in future extensions of the framework. Specialized modern multiprocessor machines could, for example, process multiple transactions of different clients synchronously. Stress tests have to be conducted to test if the conflict detection and resolution modules are still working efficiently if the server system serves a large number of clients concurrently.

To increase the overall throughout of the server system, conflict resolution should be delegated to different machines trying to handle possible conflicts detected. This could be easily implemented by defining conflict handlers that are actually web services running on different machines. Web services are becoming increasingly popular and are platform independent.

8. References

[1] George Coulouris, Jean Dolimore, Tim Kindberg, “Distributed Systems Concepts and Design”, third edition, Addison Wesley, 2001

[2] Maria A Butrico, Henry Chang, Anthony Cocchni, Norman H Cohen, Dennis G Shea, Stephen E.Smith, “Gold Rush: Mobile Transaction Middleware with Java-Object Replication”, Proceeding of the Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS), June 16-19, 1997, Portland Oregon pp 91-101

[3] David Starobinski, Ari Trachtenberg, Sachin Agarwal “Efficient, PDA Synchronization”, Department of Computer Engineering, Boston University, 2002

[4] Nuno Preguica, Jose Legatheaux Martins Miguel Cunha, “An SQL-based database combining conflict avoidance and conflict resolution”, Department de Informatica Universidade Nova de Lisboa, Portugal, 2002

[5] Mahadev Satyanarayanan, James J.Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel , David C. Steere, “Coda: A Highly Available File System for a Distributed Workstation Environment”, IEEE Transactions on Computers, Vol 39, No.4 April 1990

[6] James J. Kistler, M. Satyanarayanan, “Disconnected Operation in the Coda File System”, School of Computer Science, Carnegie Mellon University, Pittsburgh USA, 1991

[7] Andrew S. Tanenbaum, Maarten van Steen, “Distributed Systems”, Prentice Hall 2003, ISBN: 0131217860

[8] Nuno Preguiça, Carlos Baquero, Francisco Moura, J.Legatheaux Martins, Rui Oliveira, Henrique João, J.Orlando Pereira, Sérgio Duarte, “Mobile Transaction Management in Mobisnap“,Departamento de Informática, Universidade do Minho, Departamento de Informática, FCT, Universidade Nova de Lisboa, Portugal, 2000

[9] Robert Sedgewick, “Algorithms in Java Third”, third edition, Addison-Wesley 2003, ISBN 0-201-36120-5

9. Bibliography

[1] K. Segun, A.R. Hurson, V. Desai, A. Spink, L.L. Miller "Transaction Management in a Mobile Data Access System", Pennsylvania State University, Iowa State University, 1998

[2] Sachin Kumar Agarwal, "DATA SYNCHRONISATION IN MOBILE AND DISTRIBUTED NETWORKS", Master Thesis, Boston University, College of Engineering, 2002

[3] San Yih Hwang, "On Optimistic Methods for Mobile Transactions", Department of Information Management, National Sun Yat-Sen University, Taiwan, Journal of Information Science and Engineering, 16, 535-554, 2000

[4] M. Satyanarayanan , "Mobile Information Access", School of Computer Science, Carnegie Mellon University, Pittsburgh USA, 1996

[5] Butler Lampson, "How to Build a Highly Available System Without a Toolkit", 1995

[6] Rich Kong, Stephen Lau, Daniel New, "Mobile Data Access, Disconnected and Partially Connected Aproaches" , Department of Computer Science, University of California, San Diego, March 5, 2001

[7] Sanjay Kumar Madria, Bahrat Bahrgava, "Transaction Model for Mobile Computing", School of Computer Science, University Sains Malaysia, Department of Computer Science, Purdue University, West Lafayette USA, 1998

[8] James J. Kistler, M. Satyanarayanan, Lilly B Mummert, Maria R. Ebling, Puneet Kumar, Qi, Lu, “Experience in a Disconnected Mobile Environment”, School of Computer Science, Carnegie Mellon University, Pittsburgh USA, 1993.

[9] Anthony D. Joseph, “Mobile File Systems (Disconnected Operations)“, Lecture Berkley University, 2000, USA

[10] Sidney Chang, Dorothy Curtis, “An Approach to Disconnected Operation in an Object-Oriented Database“, MIT USA, 2002

[11] Ayse Yasemin SEYDIM, “AN OVERVIEW OF TRANSACTION MODELS
IN MOBILE ENVIRONMENTS“, Department of Computer Science and Engineering
Southern Methodist University, Dallas, USA, 1999/2000

[12] Puneet Kumar, “Coping with Conflicts in an Optimistically Replicated File System”, School of Computer Science, Carnegie Mellon University, Pittsburgh USA, 1990
[13] Robert Sedgewick, “Algorithms in Java Parts 1-4”, Addison Wesley 2003, ISBN: 0-201-36120-5
[14] Fabio Arciniegas, “XML-Developer’s Guide”, Franzis’ 2001, ISBN:3-7723-7703-3