A Context-Oriented Synchronization Approach

Dirk Draheim
Software Competence Center Hagenberg
Softwarepark 21
4232 Hagenberg, Austria
dirk.draheim@scch.at

Christine Natschläger
Software Competence Center Hagenberg
Softwarepark 21
4232 Hagenberg, Austria
christine.natschlaeger@scch.at

ABSTRACT

Synchronization gained great importance in modern applications and allows mobility in the context of information technology. Users are not limited to one computer any more, but can take their data with them on a laptop. Two common architectures have been developed recently, the *Data-Centric Architecture* as well as the *Service-Oriented Architecture*. This paper compares two existing technologies for the implementation of a mobile client and introduces a new approach, developed based on the requirements of a major insurance company, the *Context-Oriented Architecture*. This approach allows detection and resolution of conflicts within the context in which the objects were changed, while still ensuring data correctness and consistency. Therefore two new synchronization concepts are introduced: the *synchronization of complex objects* and *dialogue-sensitive synchronization*. An application implementing this approach has been realized and successfully deployed.

1. INTRODUCTION

The *context-oriented synchronization approach* introduced in this paper has been developed for the *PreVolution* project, executed by the *Software Competence Center Hagenberg* (SCCH) and the *Institut Fuer Anwendungsorientierte Wissensverarbeitung* (FAW) on behalf of the *Austrian Social Insurance Company for Occupational Risks* (AUVA).

Approximately 3 million employed people and 1.3 million school children and students are by law insured by this company. The AUVA takes care of victims of occupational accidents and diseases, a major goal therefore being the prevention of such accidents and diseases. This is the domain where PreVolution is settled.

The aim of PreVolution is to support consultants visiting the companies for advice and physical examination of their employees. The consultants need to synchronize data available at the AUVA onto their laptops to extend and modify the data at the company site. Since several consultants can visit the same company simultaneously conflicts are possible and should be presented to the consultant for resolution. Additionally, the synchronization process has to perform several tasks like business processes during synchronization, changing data as well as unique-constraint violation recognition. Based on these requirements, using a standardized synchronization concept was not possible and an own synchronization approach had to be implemented.

Therefore, this paper covers the challenges of implementing an own synchronization process and contributes two new synchronization concepts:

- Synchronization of Complex Objects
- Dialogue-Sensitive Synchronization

After an introduction of these concepts in Chap. 2 and a comparison with other research work in Chap. 3, we will discuss a *data-centric* as well as a *service-oriented synchronization architecture* in Chap. 4, followed by the new approach of a *context-oriented architecture*. Chap. 5 will describe the *implementation* of a *context-oriented synchronization* in detail. Chap. 6 will present two existing Microsoft technologies for the implementation of a mobile client. This will be the *Smart Client Offline Application Block* on the one hand and the *Synchronization Services for ADO.NET* on the other. Finally, Chap. 7 will present the most important *lessons learned*.

2. NEW SYNCHRONIZATION CONCEPTS

In its first section, this chapter focuses on the definition of the term *conflict* in order to follow up with a description of the new synchronization concepts. The second section introduces the *synchronization of complex objects* followed by the *dialogue-sensitive synchronization* in the third section.

2.1 Conflicts Defined

A conflict can only occur when two databases have a copy of a complex object CO with write-permission and the complex object is changed on both sides. A complex object consists of objects that conceptually belong together in the sense of real-world modelling. For more information on complex objects see [1].
CO_s complex object on the server; consists of data records from different tables $A \ldots Z$ called sub object $aA \ldots aZ$ with aA being the root sub object.
CO_1 Copy of CO on client 1.
CO_2 Copy of CO on client 2.
t_{d1}, t_{d2} point in time when CO_1, CO_2 are created/downloaded.
t_{c1}, t_{c2} point in time when sub object aI of CO_1 and sub object aJ of CO_2 are changed $\Rightarrow CO_1'$ and CO_2' with $t_{c1} \neq 0$ and $t_{c2} \neq 0$ (changes occurred on both sides).
t_{s1}, t_{s2} point in time when CO_1', CO_2' are synchronized.
t_{cs} point in time when complex object is changed on server.
Precondition:
$t_{d1} \leq t_{c1} \leq t_{s1}$ as well as $t_{d2} \leq t_{c2} \leq t_{s2}$

VLDB '08, August 24-30, 2008, Auckland, New Zealand

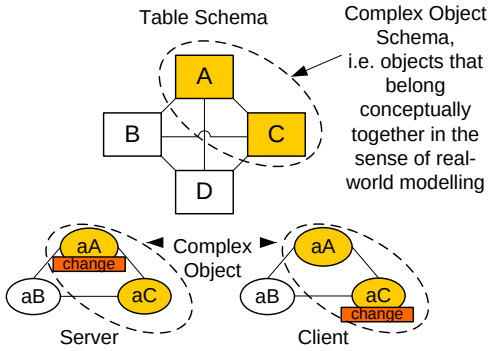

Figure 1: Synchronization Concept for Complex Objects

Assumption:
$t_{s1} < t_{s2}$ which means client 1 synchronizes before client 2
$\Rightarrow t_{cs} = t_{s1}$ (when client 1 synchronizes, its version is saved on the server).
A conflict will occur if:
$t_{d2} < t_{s1}$ (and complex object is changed on both clients (see precondition)).
Conflict will be detected at time t_{s2} when trying to set t_{cs} again without having seen the former version.

The detection of a conflict is possible by using *number ranges* for unique identification (see Chap. 5.3). Every client as well as the server has an own number range identifying all data records *uniquely*. As every sub object, the root sub object of a complex object has a unique ID. CO_1 and CO_2 are copies of the same complex object if the ID of the root sub object is identical.

The allowed operations are *Insert*, *Update* and *Delete*. Delete is only allowed in a few tables which are *checked out* (see Chap. 5.5) by the client. The conflict scenarios *Update-Delete* and *Delete-Delete* are therefore not possible. For the deletion of objects and the synchronization afterwards the concept of SyncServices (see Chap. 6.2) is used and a *tombstone table* maintains all deleted records. Through the synchronization process the entries in the tombstone table are sent from client to server, executed on server side and stored in the server tombstone table. When another client downloads data it will receive the new entries from the tombstone table and can delete the records locally.

Additionally, an *Insert-Insert* conflict is not possible either. Since every client and server has an own number range, the IDs will not be violated during synchronization.

The only possible conflicting operation is *Update-Update*.

2.2 Synchronization of Complex Objects

The *synchronization concept for complex objects* is shown in Fig. 1. For example, Table A may contain the zip code and city of an address. Table C contains the street name and street number. The objects conceptually belong together and form a complex object address.

Objects that conceptually belong together in the sense of real-world modelling are treated as one complex object so that conflicts can be detected even if data records from different data tables have been changed. A normal data replication approach would not detect the conflict and merge the data without displaying it to the user. Our approach offers a solution in this problem. Developers can specify which object classes form a complex object class. This information is then exploited at synchronization time.

There are three reasons why synchronizing complex objects can be a better approach:

- The *data model can be normalized* and does not have to consider dependent data. For example, an address can be split up in different tables with the street name in one table and the zip code in another table. If one user changes the street name of a person and another user changes the zip code, a merge can lead to a nonexistent address and is therefore undesirable. A conflict detection is necessary.
- The *data model can change*, nevertheless the same conflicts are detected. For example, the data model changes and splits up the person in two different tables. A data replication approach will detect two different conflict types based on the tables instead of one conflict for the entire person. When synchronizing complex objects the conflict will always be deteced within the entire person.
- *Detecting conflicts on objects* instead of data tables is more understandable for the user. For example, the complex object contact person has data from five different tables and is displayed to the user in one dialogue. If the user changes data on both, the client and the server he expects a conflict independent of the specific data records he has changed. In a data replication approach the user can change two different values and whether or not he will run into a conflict depends on the question if the two changed values are physically stored in the same table. Eventually, this might be confusing for the user.

Since *contact person* will be often used as an example for a complex object it will now be explained briefly. A *contact person* is a person being the contact person for a concrete company. One person can work for several companies and can therefore be contact person for more than one company. A contact person is a complex object containing of data from five different data tables. The data table *person* contains information about the person itself like name, title and birth date, the data table *contact person* provides information related to the company e.g. the e-mail address and telephone number of the person in the company, the data table *contact person function* contains the function of the contact person in the company e.g. CEO or secretary and the last data table is *address* which is connected to the person providing the private address and to the contact person containing the business address.

2.3 Dialogue-Sensitive Synchronization

The *dialogue-sensitive synchronization concept* is shown in Fig. 2. It offers a better decision support in the conflict resolution domain. The dialogue context where the change has occured is saved and presented to the user in the conflict display.

In Fig. 2 class A may stand for a contact person and class B may stand for a person. One physical person can be contact person for two different companies. The first user changes the person in the context of contact person A1 on the server whereas the second user changes the same person in the context of contact person A2 on the client. The dialogue-sensitive synchronization concept allows the user to see the conflict in the *context* of the contact person he has changed, so the server version of the conflict shows the person with contact person A1 and the client version shows the person with contact person A2.

In this paper, ”context” refers to the business logic view of objects rather than to the data view (e.g. person and contact person are two different tables but in business view they form one complex object).

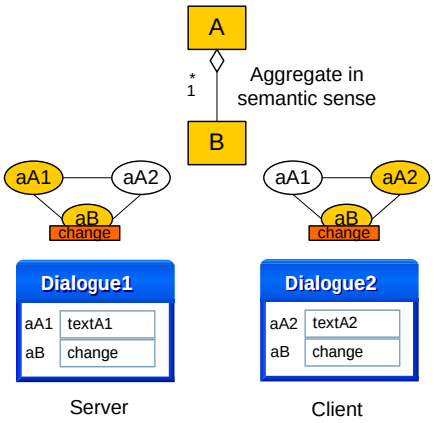

Figure 2: Dialogue-Sensitive Synchronization Concept

This concept is not limited to the same complex object. The first user can change a person in the context of the complex object patient, whereas the second user changes the same person in the context of a contact person.

3. RELATED WORK

Several papers have studied the issue of developing *synchronization for mobile environments* (e.g. [2, 3]). To improve synchronization, information about *data provenance* is needed, which is covered in detail in [4] and [5].

In [4] Foster and Karvounarakis studied provenance in the area of data replication for different devices. Replicas have to be transformed for different devices, which leads to difficulties in view update and maintenance.

In [5] Buneman et al. investigate data provenance and differ between "Why" and "Where" provenance. Data provenance describes where data comes from and the process in which the data was created or changed. "Why" refers to the source data that influenced the existence of the new data and "Where" identifies the origin location of the data.

In [6] Cui and Widom studied lineage tracing for data warehouses. During the integration of an operational data source into a data warehouse, source data is typically transformed. Data lineage covers the problem of tracing the derived data items to the original source items.

The process by which the data arrived in the database is also important in our approach in order to identify the complex object in which a data record was created or changed. Additionally, in a simple way this paper introduces a kind of "Who" provenance, where "Who" stands for the user who made the change in the database. This makes it possible to display the users, who made the changes that lead to a conflict. Similar to [4] the information is stored as metadata in the database in both cases. This is necessary to support the user with conflict resolution.

In [7] transaction processing techniques are described and used to monitor, control and update information. Transaction processing keeps a database in a consistent state by completing all transactions successfully or rolling them back otherwise. The book therefore covers fault tolerance, concurrency control as well as recovery and rollback. Keeping the database in a consistent state, identifying autonomous operations and rolling back transactions in case of an error is a demanding topic in most synchronization approaches.

In [8] Lee et al. studied data synchronization in mobile environment with focus on *conflict resolution*. For this implementation, SyncML was selected. Aspects of the synchronization are the usage of *Global Unique Identifiers* GUIDs to uniquely identify a data record and the usage of a *change log* which stores all changes that have to be synchronized. The advantage of a change log is a better performing synchronization since the single synchronization steps do not need to be reconfirmed and the execution process can be optimized locally. The aim of their approach is an *automatic synchronization* which needs a *conflict resolution policy* like originator-win, recipient-win, client-win, server-win, duplication or recent-data-win. From all these possible conflict resolution policies the recent-data-win policy has been selected. An automatic conflict resolution alleviates synchronization for the user and makes sense when all changes are made by the *same* user on different devices. However, when the changes are performed by *different* users, using a policy is a problem, especially when the policy decides that one change will overwrite the other. In contrast, our paper is based on the assumption of a mobile environment with many different users that change the same data in parallel, and it therefore offers an approach that supports manual conflict resolution.

Other related work about *replication of data for mobile environments* can be found in [9, 10]. In [9] Ratner et al. identified requirements for replication in a mobile environment. In [10] Barbara and Garcia-Molina studied dynamic replicated data management algorithms for generating and migrating replicated copies. Additionally, several different replication approaches for mobile systems are compared in [11]. Although a data replication approach might make things easier, there are some disadvantages to it, as well as requirements it can not fulfill. Chap. 4.1 will elaborate this fact.

4. SYNCHRONIZATION ARCHITECTURES

In this chapter two existing approaches for exchanging data between client and server are illustrated and evaluated. The first approach is the *Data-Centric Architecture* followed by the *Service-Oriented Architecture*. Since both approaches do not fit the requirements, a new architecture approach, the *Context-Oriented Architecture* has been developed and is discussed in Chap. 4.3.

4.1 Data-Centric Architecture

In a *Data-Centric Architecture* shown in Fig. 3 the database of the server is *fully or partly replicated* to the client database and data differences are merged. Changes can be tracked and conflicts can be detected.

Oracle supports synchronization between two Oracle databases with conflict detection [12]. However, the conflict resolution is based on simple rules like client-wins, server-wins or a custom programmatic resolution.

The advantages of this approach are that a lot of research has been made in this area and that well tested solutions are available. The disadvantages are that for this approach both databases must be *compatible*.

Additionally, the server database must be *reached directly* by the client. This can be acceptable when the server database is only exposed to a secure intranet but might be a security risk if the clients access from the Internet as required in the approach described in this paper. Moreover, a conflict resolution policy will *automatically override changes* of one user. Even if there would be the possibility to display the conflict and let the user decide, a conflict would only be displayed based on the *actual table* but not on the *context* in which the object was changed. See Chap. 5.6 for more details on context-oriented conflict detection and resolution. Another disadvantage of Data-Centric Architecture is the difficulty of *replicating*

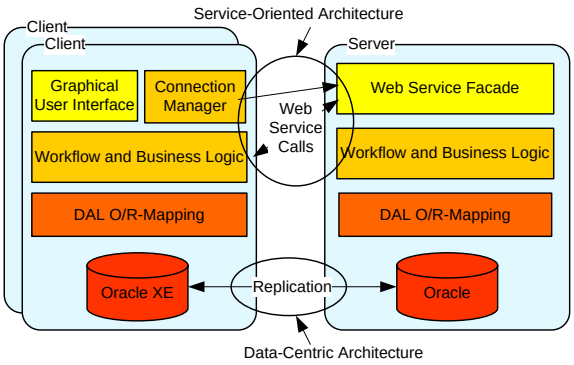

Figure 3: Data-Centric vs. Service-Oriented Architecture

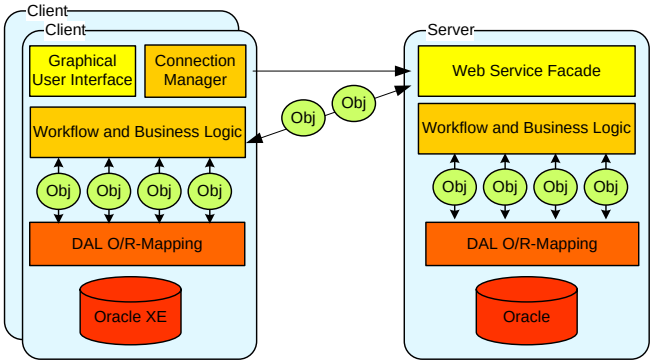

Figure 4: Context-Oriented Architecture

parts of a table based on objects as discussed for the SyncServices in Chap. 6.2. The required where-statements will be quite complex. Last but not least it is sometimes necessary to change data during replication, e.g. check data out and save it without a lock on the client or load data and save it read-only on the client. After considering these issues, a data-centric approach was not the right choice to fulfill all requirements.

4.2 Service-Oriented Architecture

The goal of *Service-Oriented Architecture* (SOA) (also shown in Fig. 3) is *loose coupling* between interacting components. Therefore, a service, which can be a *Web Service*, is offered on the net. The business logic on the client calls the server-sided Web Service. The data model of client and server can be different and in addition, the client is responsible for conflict detection and resolution, which requires custom-implemented conflict handling. Advantages of SOA are the *independence of the database*, so databases from different vendors can be used, as well as the possibility to provide better security mechanism for the server database. A disadvantage of SOA compared to Data-Centric Architecture is *worse performance*. However, the concept of synchronization over Web Services is also used in the Context-Oriented Synchronization Approach.

4.3 Context-Oriented Architecture

As both described architectures are not capable of detecting and resolving a conflict based on the context, a new approach, the *Context-Oriented Architecture*, is developed and shown in Fig. 4. Similar to the *Service-Oriented Architecture*, this approach uses a *Web Service* for communication with the server. The synchronization however is based on *objects* which are created using an O/R-Mapping tool. Every object has data from the database which is not only composed of a single relational record but several records across many tables. Client and server have relational databases with an additional table to maintain the metadata of the objects and save the context of changes.

Compared to the *Data-Centric Architecture*, the data model on client and server has to be similar, to be able to create compatible objects. However, a Data-Centric Architecture disposes of only basic conflict handling capabilities and does not support business processes during synchronization. As an example in PreVolution it is a requirement, that an offline created company has to be forwarded to the supervisor for approval during synchronization.

The Context-Oriented Architecture is similar to the Service-Oriented Architecture as both use Web Services for communication and benefit from loose coupling. Still, Service-Oriented Architecture is defined as requiring independent and committed transactions, which is not the case in our approach. The synchronization requires several dependent Web Service calls.

The advantage of the Context-Oriented Architecture is that as it is based on objects, it is also possible to *synchronize defined subsets* of the database and still guarantee *consistency*. This is an important feature as illustrated in the following example: Some tables on the server-sided database are insert-once restricted and data from these tables can be kept offline and changed several times until the user releases and commits the data. Nevertheless it is possible to synchronize the rest of the data in between, providing better data safety and performance through partial upload.

5. CONTEXT-ORIENTED SYNCHRONIZATION

This chapter describes the *implementation* of the *Context-Oriented Synchronization Approach* in PreVolution in detail. The first section gives an overview of the *architecture*, the second briefly describes the *Genome* O/R-Mapping tool and the third section explains *data model* and *number ranges*. The fourth section deals with data *upload* from client to server, whereas the fifth section discusses the opposite direction, *downloading* data from server to client. The sixth section covers the demanding topic of *conflict detection and resolution* and the last section elaborates on some *challenges* encountered while implementing the Context-Oriented Synchronization Approach.

5.1 Architecture

This section describes the *architecture* of PreVolution, as displayed in Fig. 4. On the client side *Oracle XE* is used as local database, while *Oracle Database 10g* is used on the server side.

In a previous implementation *Microsoft SQL Desktop Engine* (MSDE) was used on the client, but incompatibilities between the two databases were encountered (e.g. Unique in SQL allows only one NULL value, whereas Oracle can have several NULL values, or slightly different data types between the two databases) so MSDE was exchanged for Oracle XE. However our new synchronization approach does not need two databases from the same vendor, it works with MSDE as well.

Both, client and server, use *Genome* (see Chap. 5.2) as Data Access Layer and O/R-Mapping tool.

The *Business Logic* (shown in Fig. 5) is identical on client and server, so there is no difference between working online or offline. The client uses the ILogic Interface and depending on the client being online or offline, the specific instance of ILogic is either LogicToServer or LogicToDatabase. When being online, LogicToServer

is called, the call is forwarded to the Web Service Façade and after calling a Controller the logic in LogicToDatabase is executed, which queries the server-sided Oracle database. When the client is offline, LogicToDatabase is called directly and queries are executed on the local Oracle XE instance.

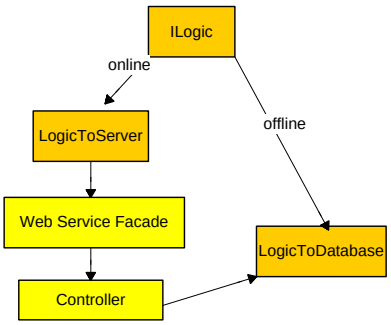

Figure 5: Business Logic

Finally the client provides the graphical user interface and the connection manager from the Smart Client Offline Application Block (see Chap. 6.1), whereas the server includes a Web Service Façade.

5.2 O/R-Mapping with Genome

Genome [13] developed by *TechTalk* is used as *Data Access Layer*. Genome is an O/R-Mapping tool for .NET and supports Oracle, Microsoft SQL Server and IBM DB2 as databases. Genome supports lazy loading (objects of a result set or attributes of an object are not loaded until they are needed) as well as *optimistic* and *pessimistic locking*.

Optimistic locking allows concurrent access, however when several clients try to commit changes, only the first client commits successfully and the other clients receive an exception. The disadvantage of optimistic locking in Genome is that an additional *version field* is necessary in every database table. As this version field has to be set by every application updating a row in the database, optimistic locking was not possible in the approach described in this paper since other applications use the same database.

The advantage of *pessimistic locking* is that the client knows upfront that data is locked and that it will not lose any changed data. In the present approach pessimistic locking is used and data will only be committed once the lock is released. Additionally the communication with the database is executed in a context which differs between *ReadOnlyContext* for read-only and *ShortRunningTransactionContext* for read/write.

The description of the data model in Genome is XML-based. For each table an Entity is created and these Entities are combined to Objects. There are two different kinds of objects: the first are the normal Genome-Objects, which are high-performance, but can not be transmitted over a Web Service. The second type is the Data Transfer Objects (DTOs), which are defined through Genome Views and can be transmitted over a Web Service. A disadvantage of Genome is that it only offers methods to serialize the Entities based on a View into a DTO but not to deserialize them on the other side.

Supported query languages are Object Query Language (OQL) and LINQ, which is new and was not supported until October 2007. Although Genome is an important and very beneficial tool in PreVolution, it has no mechanism for either *synchronization* or *conflict handling*.

5.3 Data Model and Identification

As common in many projects, the database on the server is not only used by PreVolution but also by other existing applications inside the AUVA. Therefore changes in the existing data model are difficult. *Global Unique Identifiers* (GUID) as used in [8] would have been the ideal method for data record identification. However, every table has a 64-bit long number as a primary key. Another approach was using *negative IDs* on the client for new records and replacing them later during uploading, but this is complex and costly and conflicts would have been possible on the server with new records from other applications. The best-fitting approach was using *number ranges* on server and client side. A new client without a number range, requests the number range from the server and all created records on the client receive an ID from this range. All used ranges are maintained in a database table on the server.

Every table on client and server has eight *additional columns* containing user and date information in columns *CreatedBy*, *DateCreated*, *ModifiedBy*, *DateModified*, *LockBy*, *LockDate*, and *LockType* for recognizing changes and *SyncDate* for the synchronization. The three primary columns are *DateModified*, *LockType* and *SyncDate*. Whenever a change occurs *DateModified* is set to the current date. *LockType* allows checking out data from the server or save read-only data on the client. Every lock is saved in this column. Very important for the synchronization is the client-sided *SyncDate* column, which is not populated on server-side. It contains the date at which the record was last loaded from the server. Microsoft SyncServices (see Chap. 6.2) have less but similar columns for maintaining changes.

Another aspect is the *SyncTable*. This table enables Context-Oriented Synchronization and allows context-based conflict resolution by saving all objects. Besides the *ID* this table provides the columns *ObjectType*, *ObjectReference*, *conflicts* (bool), *SyncCreatedDate*, *SyncModifiedDate* and *LastSyncDate*. When downloading a contact person with trailed records from the tables *person*, *address* and *contact person functions*, one entry would be inserted in the *SyncTable*. *ObjectType* would be contact person, *ObjectReference* the ID of the contact person in the contact person table and *conflict* would be false. *LastSyncDate* and *SyncModifiedDate* would be the same date, which has been transmitted from the server. When the client is now offline and changes a property in the table *contact person function*, the actual date is saved in *DateModified* of the *contact person function* table and additionally in the *SyncModifiedDate* of the *SyncTable* for the concrete complex object contact person. When synchronizing the changed data can be extracted from the *SyncTable* by comparing *SyncModifiedDate* with *LastSyncDate*, and if a conflict occurs in the function, the conflict can be resolved within the context of the contact person. For more information on conflicts see Chap. 5.6.

The next change in the data model is a server-side table called *ReplicationTable*.When the client starts synchronization, the ID and the name of every complex object (e.g. person) that has to be transmitted to the client is stored in this table. The first advantage is that packages of e.g. 100 persons can be requested by the client and stored in the local database. It is not necessary to send all data at once. The second advantage is that the client can abort the synchronization process after each package and, for example, continue the next day. In this case the *ReplicationTable* is updated with data that changed in the meantime, and the client can continue the synchronization process.

The data models on client and server are largely the same, although some tables are only used by the client or by the server. However there is one *additional role* on the client which has the right to delete records from all tables. No role has the right to

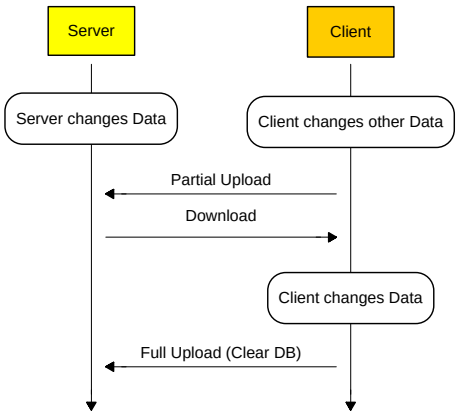

Figure 6: Synchronization Flow

delete records on the server.

5.4 Upload

As already mentioned before, *Uploading* in this paper characterizes the upload of client data to the server. There are two different types of uploads, namely *full upload* and *partial upload* (see Fig. 6). Both identify all offline changed data and send them to the server. If conflicts occur, both will *detect the conflicts* and ask the user to resolve them. However, the partial upload checks if there are offline changed data available before starting the upload, and locks on the server will not be removed. The full upload does not check for changes, it will always start the upload and if nothing has changed, at least the locks will be removed on the server.

The *partial upload* is used *implicitly* when the user wants to download data. The implicit upload before downloading is necessary, because only the upload can detect conflicts.

The *full upload* is started *explicitly* by the user and after sending all changed data to the server and resolving potential conflicts, the local database will be largely cleared, except of a few tables, which need to be always available offline and are therefore never cleared.

There is one more aspect that needs to be mentioned, because it is one of the challenges described in Chap. 5.7. Addresses are standardized and allowed to exist only once in the database. When the user creates offline a new address for a company and this address already exists on the server, during upload the objects loaded from the client are changed to point to the existing address on the server. The existing address is sent back to the client and stored in the database. All objects are changed to point to the existing address from the server and afterwards the client-sided address is deleted.

5.5 Download

Downloading refers to the download of data from server to client. A download can only occur after a partial upload, so it is not possible to run into conflicts during downloads.

There are three different *types* of data to be downloaded:

- The first group is data that are downloaded *read-only*. The existing workflow as well as old orders and documents belong to this group.
- The second group consists of data that is *checked out* from the server, so the server gets a lock and the data can only be changed on the client. For example, the current order and the route belong to this group. No other user should change the same order at the same time. In these two groups conflicts are not possible, because data are only changed on one side.
- However, there is a third group which includes data that can be *changed on both sides*. Contact persons, patients or economic domains belong to this group.

If the user downloads again, the client-sided *LastSyncDate* is compared with the server-sided *ModifiedDate* and changed data is downloaded again.

If the user has not yet downloaded or performed a full upload before, and the connection breaks, he can still work offline. There are data, which are always available offline, and these data are enough for working with the application.

5.6 Conflict Detection and Resolution

As already mentioned, conflicts are only detected during upload. The client sends its changed data and the *LastSyncDate* to the server, and if the server has changed the same data, a conflict is detected.

Conflicts are detected and displayed *context-based*, this means that if the client changes the address of a person from Main Street 3a to Main Street 3b and another user changes the address of the same person to Main Street 3c, a conflict resolution without context would display Main Street 3b versus Main Street 3c. The user would not be able to decide, because he would not know why and for whom the address was changed. It would not be possible to determine the person, for which the address was changed, when several objects like other persons and companies point to the address too. However, the Context-Oriented Synchronization knows in which context the address was changed and is able to detect and display the conflict for the person with the address.

There are different possibilities to define a conflict. The first kind of conflict is when an *attribute* is changed on both sides and there is no possibility to merge. This would be the case in the address example above, because 3b and 3c cannot be merged. Another type of conflict is characterized by being based on a *complex object* instead of a single attribute. For example, the client can change the e-mail address of a contact person and a user on the server changes the name of the person (note that contact person and person are two different tables). Without context-oriented conflict detection it would not be possible to detect this conflict during upload, because when uploading the persons, no conflict would occur. When uploading the contact persons later, there would be no conflict either. In our approach it is possible to detect this conflict because the *SyncTable* tracks that something has changed in the context of a contact person and uploads the whole contact person as one complex object.

All conflicts are shown to the user when he is online. He sees the server version with the name of the user who changed it and his version with the later changed preselected and can decide which one to take. The server version of conflicts are regularly updated, since it is possible that another user changes the server version again. When the user resolves a conflict, it is again checked whether the server version has changed in the meantime. If it has, there is a *conflict on a conflict* and the user has to decide again which version to take.

5.7 Challenges

The first challenge in this project was the server-sided data model, which was not allowed to be changed. *Number ranges* have to be used instead of *Global Unique Identifiers* and the additional columns for *change tracking* had to be inserted into separate tables. A view then combined the two tables. As deletion on server side is

only allowed for a few tables, there have been some cases, where the ID from the server has to be taken and replaced on the client during upload e.g. for standardized addresses.

Another challenge are *unique constraints* which are not directly part of the synchronization process but problems can occur during upload. The synchronization will not treat this as a conflict but it has to know all unique constraints and if during upload one constraint is violated, the user has to be informed that he has to correct the data offline before trying to upload again.

Error handling and *recovery* was another challenge. When an exception occurs during synchronization it is important that both databases remain in a consistent state. This is achieved on the one hand by using the Genome *transaction context* as described in Chap. 5.2. Using this approach allowed using Genome's generic transaction manager instead of using the specific transaction manager provided by an underlying database. A *ShortRunningTransactionContext* is opened and all changes are defined until one commit persists all changes at once into the database. Therefore partly committed data within one context will never occur.

On the other hand all methods are designed such that they are *independent* from each other. With the help of the *ReplicationTable* described in Chap. 5.3 it is possible to download independent packages of person objects. After saving a package in the local database the client sends a *delivery receipt* for the package. If the package is already stored locally and an exception occurs when sending the receipt, the server will resend the same package. If the client does not get a *delivery receipt* from the server after uploading, it will also resend the data.

As a result of an exception it is possible that the data in the local database has *different synchronization dates*, however this is also possible when the user stops the synchronization manually and does not limit the use of the application.

The last challenge and still not resolved at the moment is the *complete rollback* of the synchronization process. One requirement demands that a rollback is always possible throughout the entire synchronization process. However, some data are stored on the server and some on the client and there are several Web Service calls in between. As the Genome transaction can not be kept open for such a long duration, this is not easily solvable. However, there are other options, for example to allow a rollback after each step during upload and download as described above so that the data are always consistent.

6. SYNCHRONIZATION TECHNOLOGIES

After describing the *Context-Oriented Synchronizaion Approach* and a possible implementation, this chapter will briefly introduce *two promising technologies* from Microsoft. The first is the *Smart Client Offline Application Block* [14] followed by the *Sync Services for ADO.NET* [15]. Both technologies provided ideas for and allow comparison with the new approach.

6.1 Smart Client Offline Application Block

The *Smart Client Offline Application Block* (SCOAB) [14] is an Application Block from Microsoft that has been published first in 2004, including best practices for the design of an architecture and for solving problems in the context of online/offline scenarios. SCOAB comprises *best practices*, *design patterns* and *examples*. Additionally, the entire code is available as well, allowing to adapt the Application Block for individual needs. SCOAB is meant to support the development of offline-capable applications. Therefore, the data are stored on the client and SCOAB takes care of the synchronization as soon as a connection is available.

SCOAB uses *DataSets* for conflict detection. When implementing a prototype in 2005, DataSets did not support conflict resolution, but basic abilities were added in the meantime. For communication with the server a Web Service is used and messages are sent through *Microsoft Message Queuing* (MSMQ). For data management *In-Memory*, *Microsoft Desktop Engine* (MSDE) or *Microsoft SQL Server* can be used. SCOAB uses several threads, one for the application, a second for the connection manager and a third for the messages. Additionally, SCOAB includes an extensive evaluation of possible security risks, as well as suggestions for testcases. SCOAB is based on .NET-Framework 1.1 but can be upgraded to .NET-Framework 2.0.

A prototype at the beginning of PreVolution was developed to evaluate SCOAB. However, SCOAB was largely not included in the final version of the project since DataSets were not sufficient. One component of SCOAB, the *connection manager*, which detects if a connection to the server is available and allows switching between online and offline modes, has been integrated into PreVolution in slightly modified manner.

6.2 Synchronization Services for ADO.NET

The *SyncServices for ADO.NET* [15] are a completely new part of the *Microsoft Sync Framework* (MSF) and *.NET Framework 3.5*, which is integrated in Visual Studio 2008. SyncServices allow four different synchronization methods:

- Snapshot
- Incremental Download (Insert, Update, Delete)
- Upload (Insert, Update, Delete)
- Bidirectional Synchronization with Conflict Handling

Microsoft SQL Server Compact Edition 3.5 (CE) has to be used as local database, which is only slightly different from *Microsoft SQL Express Edition*, but takes less space, is stored in one file and can store data up to 4 GB. The database on the server can be *Microsoft SQL Server* or another database like *Oracle* or *DB2*. SyncServices allow communication over Web Services. SyncTable objects determine the tables to be synchronized; several SyncTables can be grouped into a SyncGroup, which is always synchronized within one transaction, so a rollback is possible.

SyncServices were evaluated for approximately one month in several prototypes and demos but were not appropriate for our requirements due to the following issues:

- *Conflict resolution is not fully developed* at the moment, some conflicts have to be resolved on the server or the process has to be suspended.
- SyncServices are *completely new*, Visual Studio 2008 has to be used and documentation is rare. Additionally, Microsoft SQL Server Compact 3.5, which has some compatibility problems with Oracle, is a must on the client.
- *Changes while downloading* (e.g. setting a lock) cannot be done during the synchronization process but require a separate SQL-Statement. However, this would be detected as a change and propagated to the server during the next synchronization process.
- *Synchronization of objects is not possible*, instead tables are synchronized. The records in a table can be restricted using Select-Statements. These statements would be very extensive, because there is no need to download all two million

addresses but only those connected to a person, a contact person, a company or other objects the user selects in his download list, which likely will result in approximately 600.000 addresses. The list of IDs in the Select-Statement will nevertheless be quite large.

- *Restriction of the data amount* is only possible on the server but not on the client. A partial upload from the client to the server which offers better performance on the one hand, and allows to synchronize data with insert-once restriction on the server on the other hand, is not possible with one Sync Agent. However, using different Sync Agents leads to other problems including rollbacks. Normally, rollback of transactions is supported in an efficient manner, but not possible when using different Sync Agents.
- The last issue is that the *code is not available*, as it would be in SCOAB or a custom-implemented approach. Therefore, the code cannot be extended and necessary changes cannot be implemented.

7. LESSONS LEARNED

In this project lessons were learned regarding workflow management [16], requirements engineering [17] as well as regarding synchronization. There is a huge number of different approaches for synchronization and it is difficult to decide. Evaluation takes some time and sooner or later a decision has to be made. SCOAB and SyncServices were the main but not the only evaluated technologies, however no technology fitted and an *own approach* had to be implemented. Building an own synchronization solution is not as easy as it first seems. Synchronization is something that should work in the *background* and not bother the user until it is really necessary; nevertheless, synchronization is a *main component* of every online/offline application and is quite complex. Every possible scenario can and will happen, e.g. conflict during conflict resolution. However, implementing an own synchronization approach is possible.

Another important lesson is that also if a synchronization approach does not necessarily need two databases from the same vendor or a similar data model, it is always an advantage to do so. If it is possible to have data in the *same structure*, schema mapping can be avoided, thereby reducing complexity and effort.

The last important lesson learned is that a local database has *limits* compared to the server database. Knowing that an offline database is available may tempt to think that it would also be nice to have more and more data offline. However, an offline database has a size limit and besides the synchronization process should be kept simple and fast, so it is highly advisable to download only what is really necessary and helpful, and if some data are only needed for information porposes, they should be made read-only in order to reduce the possibility of conflicts.

8. ACKNOWLEDGEMENTS

The authors gratefully acknowledge support by the Austrian Government, the State of Upper Austria, and the Johannes Kepler University Linz in the framework of the Kplus Competence Center Program. Special thanks go to all project members of PreVolution, working and have worked in this project.

9. REFERENCES

[1] Dirk Draheim and Gerald Weber: Form-Oriented Analysis: A New Methodology to Model Form-Based Applications, Springer Verlag, 2005.

[2] Shirish Hemant Phatak and B. R. Badrinath: Conflict Resolution and Reconciliation in Disconnected Databases. 10th International Workshop on Database and Expert Systems Applications (DEXA), 1999.

[3] Alan Demers, Karin Petersen, Mike Spreitzer, Douglas Terry, Marvin Theimer, and Brent Welch: The Bayou Architecture: Support for Data Sharing among Mobile Users. Proceedings IEEE Workshop on Mobile Computing Systems & Applications, 1994.

[4] J. Nathan Foster and Grigoris Karvounarakis. Provenance and Data Synchronization. IEEE Data Engineering Bulletin, Volume 30, 2007.

[5] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A Characterization of Data Provenance. In International Conference on Database Theory (ICDT), 2001.

[6] Yingwei Cui and Jennifer Widom: Lineage Tracing for General Data Warehouse Transformations. Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01). 2001.

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

[8] YoungSeok Lee, YounSoo Kim, and Hoon Choi: Conflict Resolution of Data Synchronization in Mobile Environment. Computational Science and Its Applications - ICCSA, 2004.

[9] David Ratner, Peter Reiher, Gerald J. Popek, and Geoffrey H. Kuenning. Replication requirements in mobile environments. Mobile Networks and Applications, Volume 6, 2001.

[10] Daniel Barbara and Hector Garcia-Molina. Replicated Data Management in Mobile Environments: Anything New Under the Sun? Applications in Parallel and Distributed Computing, 1994.

[11] Astrid Lubinski and Andreas Heuer: Configured replication for mobile applications. Proceedings of the IEEE International Baltic Workshop on DB and IS, BalticDB&IS'2000, 2000.

[12] Philip Stephenson. Oracle Database Lite 10g Technical White Paper, May 2005.

[13] TechTalk. Genome - Generative Object Mapping Engine, White Paper, 2006.

[14] Microsoft. Smart Client Offline Application Block. URL, http://msdn2.microsoft.com/en-us/library/ms998460.aspx, 2004.

[15] Microsoft. Sync Services for ADO.NET. URL, http://msdn2.microsoft.com/en-us/sync/bb887608.aspx, 2008.

[16] Theodorich Kopetzky and Dirk Draheim. Workflow Management and Service Oriented Architecture. SEKE 2007, pages 749-750, 2007.

[17] Mario Pichler, Hildegard Rumetshofer, and Wilhelm Wahler. Agile requirements engineering for a social insurance for occupational risks organization: A case study. In RE '06: Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06), pages 246-251, Washington, DC, USA, 2006. IEEE Computer Society.