AALBORG UNIVERSITET

Data Synchronization Between Mobile Devices in a Multi-Project Setting

Semester Project by sw612f14

from 03-02-2014 to 28-05-2014

AALBORG UNIVERSITET

Third study year
Computer Science and Software
Selma Lagerlöfsvej 300

Title: Data Synchronization Between Mobile Devices in a Multi-Project Setting

Project period: Spring semester 2014

Project group: sw612f14

Participants:

Morten F. Baagøe
Tommy A. Christensen
Anders R. Nielsen
Toke A. Wivelsted

Supervisor:

Thibaut Le Guilly

Copies: 6

Total Pages: 114

Appendix: 4

Completed: 28-05-2014

Abstract:

The aim of the project is to develop a method for synchronizing data between mobile devices in a large multi-project setting.
The group designed two methods, one using timestamps and another using message digest, for synchronization between a single server and multiple mobile devices.
Implementation began on both approaches and criteria for comparing the two were elaborated. However, focus was shifted to a partial solution, thereby enabling access to shared data for the other groups in the multi-project.
Because of this shift and unforeseen complications in other areas of the multi-project, no final approach was completed.
Thus, leaving the question of which approach to use unanswered.

Foreword

This report was made at Aalborg University as the Software engineering bachelor project by the group sw612f14. The report was made as a part of the SW6 multi-project in the period between 3rd February and 28th of May of the year 2014. The project was supervised by Thibaut Le Guilly, who has been very helpful throughout this project.

Morten F. Baagøe

Tommy A. Christensen

Anders R. Nielsen

Toke A. Wivelsted

Summary

The aim of the project is to develop a method for synchronizing data between mobile devices in a large multi-project setting.

The report is divided into four sprints, detailing what have been accomplished during this semester. The first sprint was spent mostly on researching and prototyping the Android platform and studying the former version of the project. In the second sprint, we focused primarily on building a workable one-way synchronization between clients and server, and analyzed other existing solutions. For the third sprint we decided to start implementing another approach to synchronization based on our previous analysis. The purpose of sprint four was to conduct experiments and conclude on which approach to use, but the majority of time available was spent on bug-fixing and preparing for project delivery.

In the end, we delivered a working partial implementation in cooperation with other groups.

Contents

Part I

Introduction

1 Role of project

The purpose of this multi-project is to develop software to aid communication and learning for autistic children. Autistic children typically have difficulties with communication and social interaction. Therefore, both children and pedagogues use pictograms, which are small simple pictures that represent words, to ease communication. These are used in many ways from giving instruction for a specific action, to telling stories and for the child to make requests. The problem with pictograms is that as a physical object the pedagogues need to print out and laminate each instance of a pictogram, and as there often are cases where the same pictogram is used multiple times and places, this leads to a lot of redundant work better spent interacting with the child.

The multi-project's aim is to make a software system that replaces the physical objects while handling all scenarios where these pictograms are used. This would allow each pictogram, once created, to be reused in many different contexts simultaneously. The software system consists of many sub-modules to accommodate the different cases where pictograms are used. Some modules focus on integrating a communication-tool with help of pictograms, another module is using pictograms in games to help train the children in various skills. Another module handles the administrative part of managing and distributing the pictograms. This is a legacy project started in 2011, and have since been used as basis for bachelor projects since then.

The lifespan for the multi-project this semester was split into four time-slots called sprints. Students were split into 16 groups of about four people each, where each group would pick one module of the project to work on for the first sprint. Once a sprint completed there would be conducted a sprint review where progress on the different modules would be presented to clients and fellow students. The clients would give feedback on the different presentations when relevant. Based on their input and on the progress made last sprint the groups would then evaluate what parts needed work and distribute the workload for the next sprint. Since many of the modules are not functioning independently, a lot of communication between the groups were necessary. To facilitate this, a meeting was held once every week, to discuss the progress of each group and any issues discovered. Furthermore, a server with a wiki and forum was setup to ease information sharing.

The part of the project that we worked on, all four sprints, was the task of synchronizing each tablets local database with a master database located on a remote server. This would allow each tablet to function while being disconnected from the server and, once reconnected, to share and receive updated data.

2 Requirements and Scope

2.1 Customer Requirements

Another group in the multi-project was responsible for customer relations and wrote a requirement specification for the entire multi-project. Based on this information, our group extracted the requirements that concerns the synchronization-project. As synchronization is a complex process, exactly how to implement it was left for the group to decide. But, as a minimum, the final solution should include the following features:

- Data persistency
  - It must be possible to submit any data-changes made on a client, to the server.
  - Likewise, clients must be able to receive changes made on the server.
- The synchronization process must be able to handle and recover from unexpected crashes.
- The synchronization process should be callable from any other application in the multi-project.

2.2 Project Scope

As the synchronization-project is only a small part of the larger multi-project, we need to define clear project-borders to not overlap with other project-groups. Therefore, anything that does not solely pertain to the synchronization-process itself, is out of the project-scope. The following lists gives examples on what is part of the scope, and what is not part of the scope, respectively.

In scope

- Implementing the different steps needed for synchronizing content from the server to the clients, and vice versa.
- Making the synchronization-process available for initialization.
- Conflict and error handling, when sending and receiving data between client and server.

Out of scope

- Decide structure or size limit for database-elements used in other projects.
- Handle any security-situations where a client might manipulate or destroy data and send these changes to the server through synchronization.

  - As long as the synchronization process completes, any changes made by a client is expected to be valid.

- Implementing applications or buttons for initializing the synchronization-process.

Part II

First Sprint

1 Introduction

The purpose of this section is to give an overview of the chosen project for the current sprint. A description of the project's functionality and state will lead to a set of possible problems for the group to solve. These problems will be explained and lastly, a reasonable and feasible subset of problems will be chosen as the focus of the sprint.

1.1 State of Affairs

For the first sprint, the group chose to look at the existing *wasteland*-project, whose purpose was to synchronize data between an unknown number of clients and a single server. Furthermore, the project was responsible for managing the structure of the server's database and any related communication needed to the server.

This leads to the following major aspects of the state of the project upon delivery. The previous group had successfully created a database for the server and an application to synchronize with clients. Unfortunately the remote database wasn't set up on the server, so no one could access it. Additionally, the schema used for clients in the working solution was completely different from the schema used by all other parties in the multi-project. Furthermore, the synchronization protocol had several flaws as noted in their report [1]. For instance, a communication protocol written in c++, was created to handle requests to the server, which required users to encode their requests in a very strict format using Simple Object Access Protocol (SOAP) [2]. This communication system would on occasion crash when receiving badly formatted input. Another flaw, or limitation, was that the synchronization only supported one client. If two clients were introduced they would replace each others changes causing one of them to be lost. The entire system where actual clients are unable to synchronize with the server, is illustrated in figure 1.1.

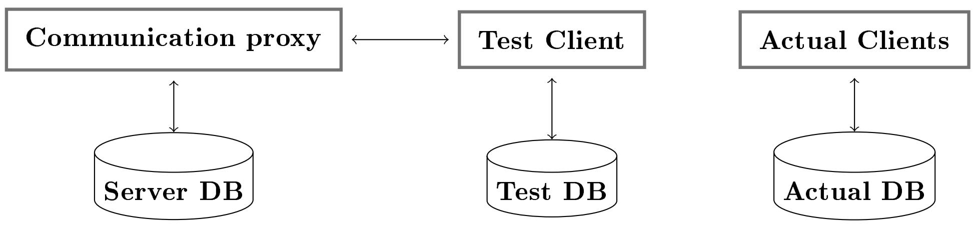


Figure 1.1: The existing system

1.2 Current Problems

The major problems relating to the project will now be listed, with the backlog from the previous project-owners report [1] available in appendix A.1.

Observed problems

- Server and client database-structure not identical
- No server-database exists.
- Faulty communication-API.
- Synchronization protocol support only a single client.

1.3 Sprint Goal

To ensure data-consistency throughout the client devices, any data changes must be distributed in full.

The current implementation is not built with the client's data-structure in mind and is therefore unable to share any data across the network.

To solve this issue, we will analyze the existing system and identify salvageable elements to implement a database-structure capable of supporting the different data.

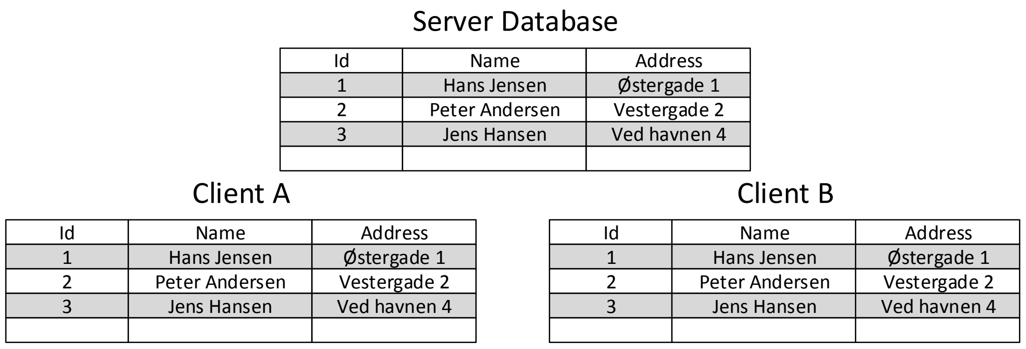
Secondly, a new system to handle distribution of data will be designed and described in detail. Thus the goals for sprint 1 are:

- Get an overview of the existing system and synchronization in general
- Setup the server database
- Make clients and server schema identical
- Become knowledgeable in working with Android Studio and other needed tools

2 Analysis

2.1 The Existing System

As mentioned earlier in section 1.1 the current way synchronization is handled is not optimal, which is illustrated in figures 2.1, 2.2 and 2.3. The problem is, that clients can overwrite each others data on the server. Whoever commits it's changes to the server last will be the winner in this backward race-condition system.



Server Database

Id	Name	Address
1	Hans Jensen	Østergade 1
2	Peter Andersen	Vestergade 2
3	Jens Hansen	Ved havnen 4

Client A

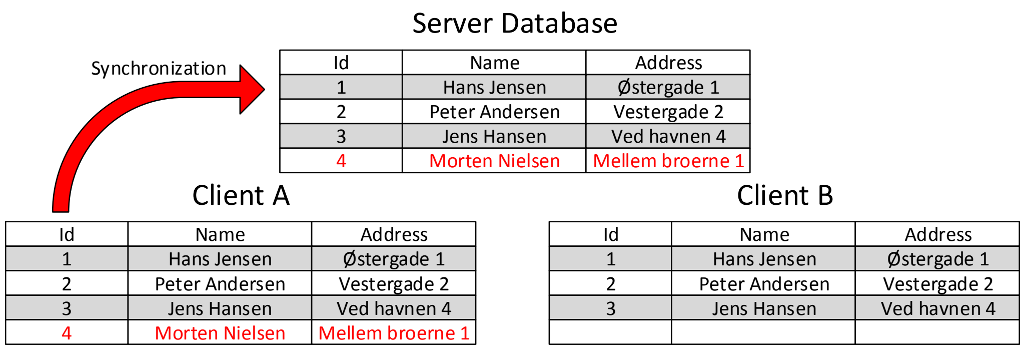
Id	Name	Address
1	Hans Jensen	Østergade 1
2	Peter Andersen	Vestergade 2
3	Jens Hansen	Ved havnen 4

Client B

Id	Name	Address
1	Hans Jensen	Østergade 1
2	Peter Andersen	Vestergade 2
3	Jens Hansen	Ved havnen 4

Figure 2.1: All 3 instances are synchronized

The problem occurs when more than one client tries to synchronize data from it's own database with the server. With the current implementation, a new row is assigned an arbitrary id without knowing whether this id is already used, so upon synchronization this would result in a conflict where the id of the newly created row would already exist, and therefore it would override the previous row since the system thinks that it is an update to the existing row. As seen in figure 2.1 the server database and both clients have the same version of the database, then when Client A has new data and tries to synchronize, see figure 2.2, the new row is inserted without problems. But when Client B contains it's own new data, the row it wants to insert already exists and as a result the server-database is overwritten with the new data from Client B, see figure 2.3.



Server Database

Id	Name	Address
1	Hans Jensen	Østergade 1
2	Peter Andersen	Vestergade 2
3	Jens Hansen	Ved havnen 4
4	Morten Nielsen	Mellem broerne 1

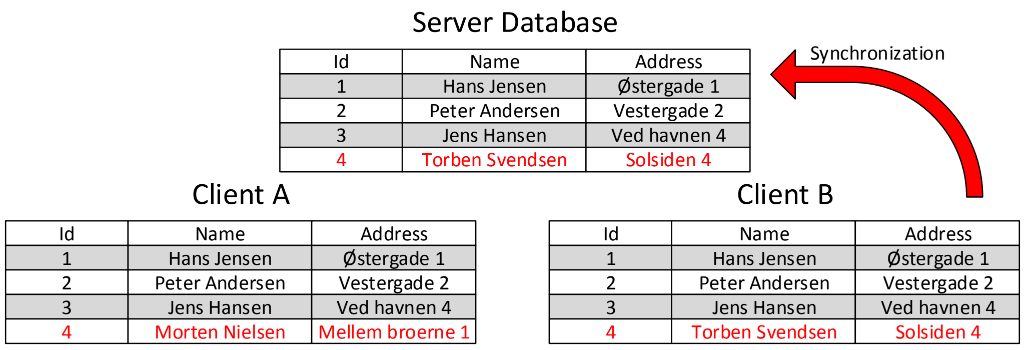
Client A

Id	Name	Address
1	Hans Jensen	Østergade 1
2	Peter Andersen	Vestergade 2
3	Jens Hansen	Ved havnen 4
4	Morten Nielsen	Mellem broerne 1

Client B

Id	Name	Address
1	Hans Jensen	Østergade 1
2	Peter Andersen	Vestergade 2
3	Jens Hansen	Ved havnen 4

Figure 2.2: Client A synchronize after adding a new row



Server Database

Id	Name	Address
1	Hans Jensen	Østergade 1
2	Peter Andersen	Vestergade 2
3	Jens Hansen	Ved havnen 4
4	Torben Svendsen	Solsiden 4

Client A

Id	Name	Address
1	Hans Jensen	Østergade 1
2	Peter Andersen	Vestergade 2
3	Jens Hansen	Ved havnen 4
4	Morten Nielsen	Mellem broerne 1

Client B

Id	Name	Address
1	Hans Jensen	Østergade 1
2	Peter Andersen	Vestergade 2
3	Jens Hansen	Ved havnen 4
4	Torben Svendsen	Solsiden 4

Figure 2.3: Client B synchronize after adding a new row

2.2 Possible Solutions

To gain an understanding of how synchronization is used today, other available systems were researched and different implementations were discussed as possible replacements to the current solution.

2.2.1 Timestamps

A synchronization system needs to be able to distinguish new data from existing data. Otherwise, too much or too little data might be sent to the requesting clients. To establish this distinctions the group found timestamps to be a possible solution.

A timestamp is a datatype which represents a specific time, e.g. "2014-03-13 08:15:00". The timestamp is typically used to specify when a record was created or changed. For the purposes of synchronization, a timestamp could be added to each record in the database and be set to automatically update it's value whenever the record changes. As the project uses two different database management systems, a description of how to handle timestamps in each now follow.

Using timestamps in MySQL
MySQL uses the statement "ON UPDATE" to specify what should happen to a timestamp once the record has been changed [3]. Therefore, the complete SQL script, as shown in List 2.1, to create a table called *Pictogram* with the attributes *id*, *name* and a timestamp *ts* becomes trivial.

```
CREATE TABLE pictogram (
  id INTEGER AUTO INCREMENT PRIMARY KEY,
  name VARHCHAR(30),
  ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

List 2.1: Timestamp settings in MySQL

The default value *CURRENT_TIMESTAMP* represents the exact time when the record was inserted into the database. Likewise, whenever the data is changed "ON UPDATE" will set *ts* to the current time. Actually, these settings are enabled by default in MySQL, so creating the desired timestamp becomes even simpler as shown in List 2.2.

```
CREATE TABLE pictogram (
  id INTEGER AUTO INCREMENT PRIMARY KEY,
  name VARHCHAR(30),
  ts TIMESTAMP
);
```

List 2.2: Trivial timestamp example in MySQL

Using timestamps in SQLite
SQLite supports setting a default value, but does not support the shorthand notation for updating a timestamp [4]. To adjust for this limitation, a trigger will need to be implemented. A trigger [5] is an operation that is automatically performed when certain events occur in the database. These events are *DELETE*, *INSERT* and *UPDATE*. An example could be to update a timestamp attribute named *ts*, whenever the attribute called *name* is updated in the same record. This trigger is shown in List 2.3.

```
CREATE TRIGGER update_pictogram_name UPDATE OF name ON pictogram
  BEGIN
    UPDATE pictogram SET ts = CURRENT_TIMESTAMP WHERE id = old.id;
  END;
```

List 2.3: Triggers in SQLite

2.2.2 SymmetricDS

SymmetricDS (SDS)[6] was found to be a viable candidate for the project-needs this semester. SDS is an open source project which focuses on synchronization for both small and large projects, primarily intended for use in multi-layered organizations. Figure 2.4 shows how a simple implementation of two client, having their own database, connects to a remote server to be synchronized. A node can act as a client, a server or even both. There are several ways to implement SDS, e.g. it can run as a background-process or be incorporated into a java application. SDS uses triggers [7] to control the synchronization process for the different nodes, e.g. clients can opt in to be notified when content of certain tables are altered.

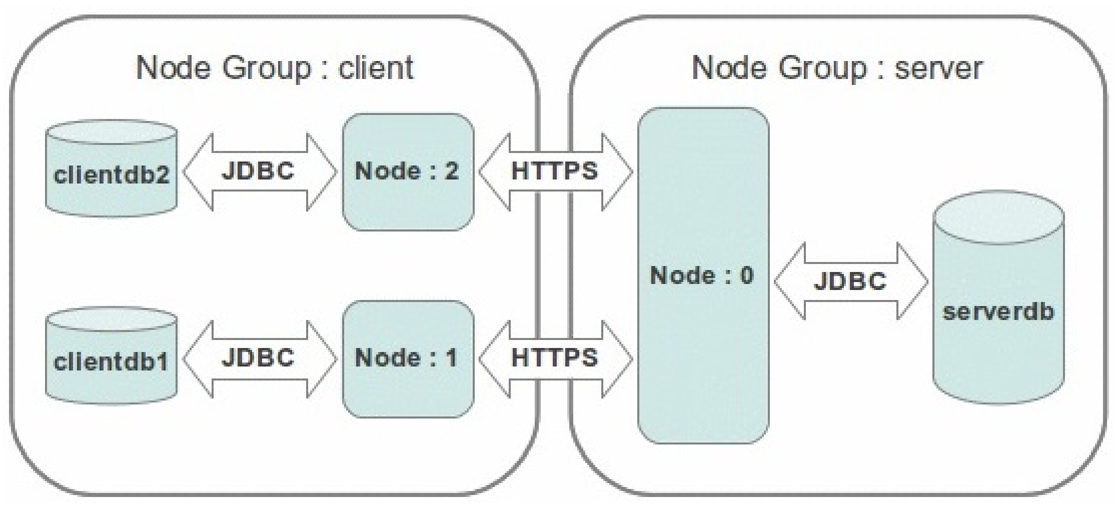


Figure 2.4: Two client-nodes connecting to a server-node over HTTPS in SymmetricDS [6]

2.2.3 Simplifying the approach

After testing SymmetricDS, discussion of how to simplify the approach began. Taking into account that clients could be disconnected from the internet for longer periods of time, any changes made on the clients, should be saved and sent to the server once a connection is reestablished.

To accomplish this, the table *DataUpdateRequests*, as shown in table 2.1 could be implemented in the clients database. The purpose of this table is to save a reference to the altered data in the clients database. E.g. if the client decides to create a new *Pictogram*, the data will be saved in the clients database and a reference will be created automatically in *DataUpdateRequests* for that particular record. The attributes of this table are id, table_name, item_id and operation.

Attributename	**Type**	**Nullable**
id	integer	false
table_name	varchar(50)	false
item_id	integer	false
operation	char	false

Table 2.1: The table *DataUpdateRequests* containing references to changes not yet synchronized

The id-attribute is the primary key, and is simply there to identify the different records. The table_name-attribute contains a string of characters representing the name of the table wherein the alteration was made, e.g "Pictogram" or "Profile". The item_id-attribute points to a particular record in the table from table_name. Lastly, the operation-attribute specifies what kind of operation was performed, i.e. *INSERT*, *UPDATE* or *DELETE* represented as 'I', 'U' and 'D' respectively.

Likewise, any changes made on the server while clients are unresponsive should be available whenever said clients become active again. Therefore, the table named *DataUpdates*, as shown in table 2.2, was designed to functions on the server exactly as the *DataUpdateRequests*-table functions on the clients. Whenever data in tables are altered, references to these alteration will be stored herein.

Attributename	**Type**	**Nullable**
id	integer	false
table_name	varchar(50)	false
item_id	integer	false
operation	char	false

Table 2.2: The table *DataUpdates* containing all data updates ever performed on the server.

The clients are now ready to receive new content, which is done partly by using the third table called *DataUpdatesPerformed* as shown in table 2.3.

Attributename	**Type**	**Nullable**
data_update_id	integer	false

Table 2.3: The table *DataUpdatesPerformed* containing all updates performed on a client

The purpose of this table is to keep track of any updates performed on a client. In short, the client looks up which update it last implemented, e.g. data_update_id 210. This attribute references the updates stored on the server in the *DataUpdates*-table. If any updates exists with a higher id than the highest data_update_id, these will be downloaded and performed in numeric sequence, i.e. 211, 212, 213 and so forth. Every time an update is performed, a new entry is inserted into the *DataUpdatesPerformed*-table referencing that particular update id.

The handling of any conflicts, pertaining to updates referencing data which the client have already altered differently or completely deleted, are unspecified at this time, but will be looked into in a future sprint. Once updates have been performed, clients will look through their *DataUpdateRequests*-table and send these data to the server. The server will then update it's own database, by first inserting or updating the content. New records will be inserted into the *DataUpdates*-table as data in other tables are altered on the server. Once the client receives an acknowledgment from the server that all data have been successfully transfered, it will delete the requests stored in *DataUpdateRequests* and any data these records references. These deleted records simply functions as temporary elements locally on the clients until they are sent to the server, as it is the servers responsibility to assign a correct id to the records and ensure data-consistency across the system. Finally, the update process is run again to ensure any changes pushed to the server and afterwards deleted locally are actually performed on the client.

Using this approach, clients could potentially be offline for example 6 months or more and one day return online and start receiving updates in the order they were performed.

As is typical for many agile projects, the design and structure of the database is prone to change during the project's lifespan. Therefore, it would be smart if structural changes performed on the server database could be distributed to clients, same as data updates. Luckily, this is an easy fix. First, a table will be added to the servers database called *StructureUpdates* as shown in 2.4. This table simply contains an id and a full SQL-command to undertake the structural changes, e.g. *DROP TABLE Pictogram;*

On each client another table called *StructureUpdatesPerformed* as shown in 2.5 will be created. The purpose of this table is to keep track of which structural updates already have been performed on a client.

Finally, to make sure every data-update can be performed, they will reference what structure-id they were created under, as shown in the updated *DataUpdates*-table in table 2.6. The clients can

Attributename	**Type**	**Nullable**
id	integer	false
command	text	false

Table 2.4: The table *StructureUpdates* contains any structural changes made on the database

Attributename	**Type**	**Nullable**
structure_update_id	integer	false

Table 2.5: The table *StructureUpdatesPerformed* contains the structural updates performed

then, when synchronizing data, check the structure_id of each data-update. If this id does not exists in the *StructureUpdatesPerformed*-table, they first need to request this structural update from the server and perform the SQL-command on the database.

Attributename	**Type**	**Nullable**
id	integer	false
table_name	varchar(50)	false
item_id	integer	false
operation	char	false
structure_id	integer	false

Table 2.6: The table *DataUpdates* containing all data updates ever performed on the server and updated to reference a structure-id

2.3 ID Handling

Id handling is the action of assuring no data gets lost or accidentally overwritten during the synchronization process due to clients using identical id's for their data. As described in 2.1, the existing system does not take id handling into account. In the following, several solutions to this problem will be presented.

2.3.1 Universally Unique Identifier

One possible solution is using a universal unique identifier (UUID) as the primary key for all data. A UUID is a 128 bit number giving a total of 2^{128} (more than 340 undecillion) unique identifiers. Different methods for generating a UUID exists, and one approach, as explained by P. Leach, M.Mealling and R. Salz in their paper *"A Universally Unique IDentifier (UUID) URN Namespace"* [8], is capable of generating 10 million UUIDs per second per machine, guaranteed to never generate an existing UUID until around A.D. 3400. This is possible by using a combination of several unique and changing value, including different timestamps and the machines MAC-address to generate the UUID.

2.3.2 Server Decision

Another approach is to store the type of action performed on a client, and send it to the server when synchronizing. The server would then decide whether to insert, delete or update a record based on the store action type. E.g. Two clients A and B create a new record and store the type as *INSERT*. Then when synchronization commences, the server can see that the data need to be inserted and creates a new record for it. This new record on the server will contain a unique id which is returned to the client who in turn updates his own id and corresponding relations.

2.3.3 Range Distribution

A third approach is to assign a unique range of ID's to each client, so every time a client adds new content, the ID is assured not to be in conflict. The amount of ID's available in the system is specified in the database schema. In the current schema identifiers uses an 32bit integer, resulting in 4.294.967.296 unique numbers being available. The assignment of sequences can be accomplished in several ways and a selection of these will be described.

Single-assignment Approach

The first approach is to assign a specific range of ID's to each client the first time they connect to the server. The first client connects and gets assigned an arbitrary number of ID's, say for example all the ID's from 1.000.000 to 1.999.999, the second client get ID's 2.000.000 to 2.999.999, the 157th client get ID's 157.000.000 to 157.999.999. This approach will support up to 4293 client and each client will be able to create one million unique records. Obviously, this approach doesn't scale well, as there is a limit on how many clients are supported and how many records each client can create. Furthermore, the approach is not very dynamic as different clients have different needs and there may not be any reason to assign one million ID's to a client who rarely or never created new data.

Multi-assignment Approach

The second approach distributes ID's on a per need basis. Whenever a client connects for the first time, it is given a pre-specified amount of ID's in a range, e.g. 500 to 1500. If the client ever is in need of additional ID's, these can be requested from the server, who will return a new range to the client. The different ranges could be stored in the client database, as shown in table 2.7.

Attributename	**Type**	**Nullable**
<u>from</u>	integer	false
<u>to</u>	integer	false

Table 2.7: The table *Ranges* showing which ranges of ID's have been granted to a client

As some data-elements will be used more than others, it makes sense to grant different tables, different ranges. Clients will for example most likely create more *Pictogram*'s than *Profile*'s, and as a result request additional ID's. But there is no need to hoard ID's for profiles, therefore, an attribute to indicate which table specific ID-ranges have been granted to, could be added to the *Ranges*-table as shown in 2.8

To control what ranges already have been distributed, the server could use a simple count-table as illustrated in 2.9, where *table_name* specifies which table this row refers to, e.g. *Pictogram*. The *counter*-attribute shows how many ID's have already been distributed and *increment* specifies how many ID's should be granted upon a request. With this approach, no client will be hoarding a lot of unused ID's, which could be used elsewhere. Furthermore, an arbitrary number of clients will be supported and if a client ever needs additional identifiers, it can be requested.

Attributename	**Type**	**Nullable**
from	integer	false
to	integer	false
table_name	varhcar(50)	false

Table 2.8: The table *Ranges* showing which ranges of ID's have been granted to which tables on a client

Attributename	**Type**	**Nullable**
table_name	integer	false
counter	integer	false
increment	integer	false

Table 2.9: The table *Ranges* (on the server) showing how many ID's have been granted on the different tables

2.4 Deletion of elements

To ensure data-consistency on multiple clients, special care is needed when deleting elements in the system. In one particular scenario, clients might not know that an element have been deleted from the server. E.g. if two clients A and B have just synchronized, and therefore contain the same data, A may decide to delete an element and tell the server to do the same through a new synchronization. The next time B wants to synchronize, the server cannot see that it needs to tell B to delete the element, as the server no longer contains it. Depending on the synchronization approach, B may see the element missing on the server and send the data needed to create it again.

To account for this, an additional attribute called *deleted* could be added to all records in the remote database. Then, instead of actually deleting records, the *deleted* value could simply be changed from *false* to *true*. This solution, however, is not particular desirable, as it will never free up space from deleted elements and leave the server filled with unused data. In future chapters, when designing a solution for synchronization, this issue will be brought up again, to see if a better solution can be found.

3 Design

In this chapter, a description of why the group deemed it necessary to replace the existing system will be provided, followed by a specification of how it will be replaced.

3.1 Problems with Existing Solution

As explained in 1.1 the current system to handle synchronization consists of a database on the server, a database on each client, a proxy-application to handle requests and an android application to submit requests, as depicted in Figure 3.1.

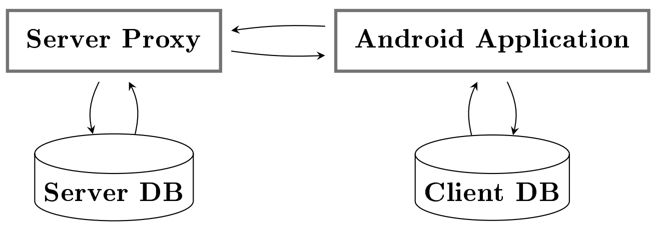


Figure 3.1: The existing synchronization system

The reasons for replacing the existing system are:

- Whenever some data-structure changes in either databases, both the Server Proxy and the Android Application will need to be updated, as redundant data-representations exists both places.
- The Server Proxy and Android application both send unencrypted messages.
- To maintain and operate the system, the developers needs extensive knowledge about the languages: c++, Java, SQL and the communication-protocol SOAP.
- Clients can overwrite each others data on the server.
- Clients can crash the Server Proxy.
- Clients must manually initialize the synchronization process.

In summary, the current system as a whole is simply not working, and the importance of certain components existence is questionable, e.g. the need for a proxy between client a server. Therefore, not only to implement a working synchronization system, but also to make maintenance easier in the future, a new system will be designed.

3.2 Designing a New Solution

With the previously mentioned limitations and flaws in mind, a design-phase began with the purpose of fulfilling the client's need for a complete synchronization system. The first step was to evaluate the importance of a Server Proxy. Several good reasons exists for having a Proxy between the servers database and any connecting clients, e.g. for security reason, the actual IP-address of the database could be hidden with a proxy. Another very valid reason to implement a proxy is to have one global decision maker. If there is two clients making changes to the same pictogram, who decides which changes to keep? Obviously each client wants to keep their version, but the proxy is better situated to evaluate any differences made and make a decision on what to do.
In the end, it was decided that a proxy would not be necessary to implement the synchronization system. Instead, minor structural database changes along with a library from where calls to both databases would be created to yield the same functionality as a proxy. These structural changes include adding a timestamp- and deleted-attribute to each table in need of synchronization, as described in sections 2.2.1 and 2.4 respectively.

Figure 3.2 illustrates the project's goal, which is a Synchronization library capable of handling all actions internally while providing a single initialization-function callable from a background-task or application running on the android device.

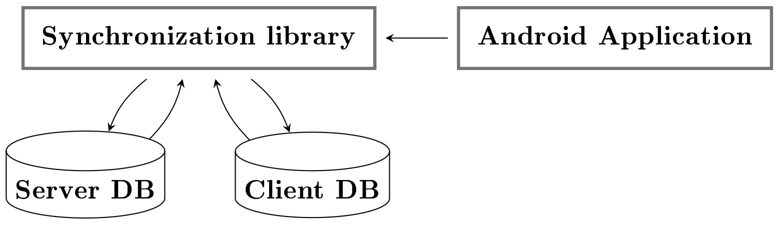


Figure 3.2: The goal synchronization system

The complete synchronization process from initialization to end is shown in the sequence-diagram Figure 3.3.

Using the abbreviations SLib for Synchronization library, AApp for Android Application, SDB for Server Database and CDB for Client Database, the different steps in the synchronization process will now be introduced. At this stage, the different steps only serve as an overview and will, in future chapters, be referenced and explained in full as they are chosen for implementation in a sprint.

Step 1) The first step is simply the AApp announcing it is ready to synchronize by calling the *Synchronize()* function in the SLib.

Step 2) Secondly, the SLib will check when the CDB was last updated, which is stored locally in the CDB, and use this timestamp to request any data from the SDB with a newer timestamp.

Step 3) As the third step, the SDB will return these new data to the SLib.

Step 4) Then, the SLib will attempt to save and merge the new data with the CDB.

Step 5) If no errors occurred, the CDB will return an *OK* and move to step six. If an error did occur, then depending on the error SLib will abort or try again by going to step two.

Step 6) Next, the SLib will request any changes made to the CDB before the timestamp previously used in step two.

Step 7) The new or altered data in the CDB is returned to the SLib.

Step 8) The SLib will then take these changes collected from the CDB and attempt to push them to the SDB.

Step 9) If the SDB successfully stores these changes, it returns an *OK* and moves to step ten. If an error occurred, the system will either abort or try again, depending on the error.

Step 10) Finally, as the last step, the SLib will save a new timestamp in the CDB to represent when synchronization took place.

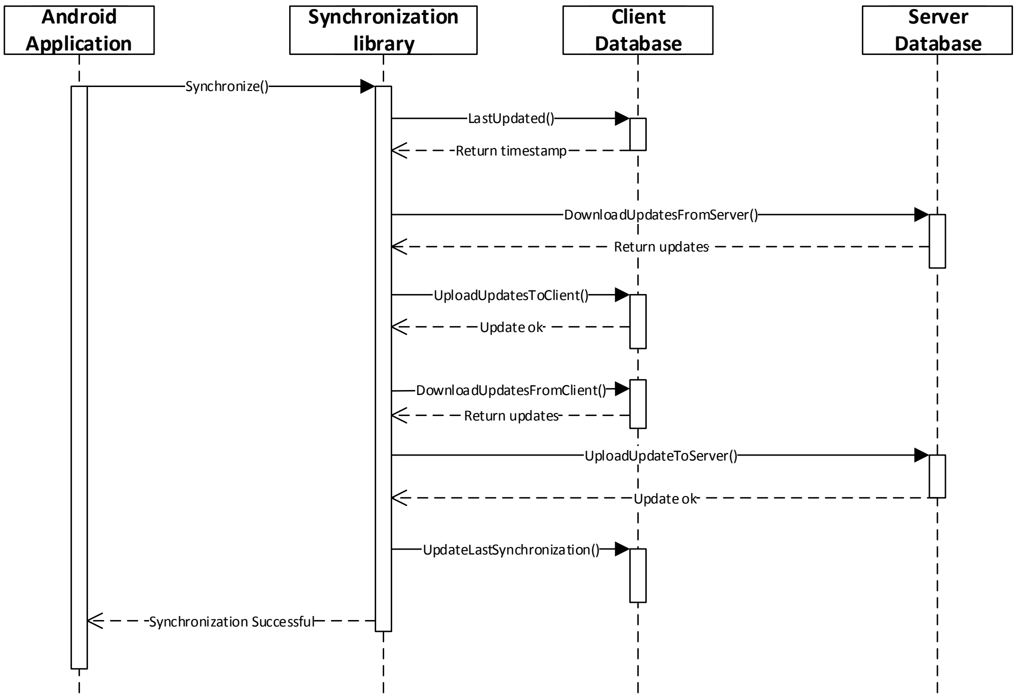


Figure 3.3: Sequence diagram for the synchronization process

4 Implementation

4.1 Proof of Concept

To better our understanding of the platform, and gather experience working with databases in conjunction with the Android framework, we created a proof of concept, for testing the feasibility of the previously described design and ideas.

4.1.1 Database Server Connection

As Android is only supplied with a standard SQL driver and library for handling the connection and transactions between the application(**app**) and the remote database, a third party library is needed to make it possible for the application work with a MySQL database.

MySQL supplies a JDBC(Java Database Connectivity) driver, called MySQL Connector/J, which makes it simple and easy to establish a connection and query a database. To connect and query a MySQL database using MySQL Connector/J only requires a few steps. First, it needs the address for the database, which needs to follow the URL standard using the following;

```
jdbc:mysql://<address to database>:<port>/<database name>
```

List 4.1: MySQL Connector/J connection string

Furthermore, it needs a username and password. A complete example for connection using MySQL connector/J can be seen in listing 4.2.

```
    private static final String URL =
        "jdbc:mysql://cs-cust06-int.cs.aau.dk:3306/test";
    private static final String USER = "sqluser";
    private static final String PASS = "sqluser";

    Connection conn = (Connection) DriverManager.getConnection(URL, USER,
        PASS);
```

List 4.2: Connect to MySQL database using JDBC driver

After the connection have been established, calls can be made to the database using a number of different ways, either as a normal SQL query sent to the server or as a call to a stored procedure on the database. Both of these can be executed as single calls or as part of a transaction, which is important in regards to the project, as it will help make sure that not only a part of the database is updated because of a failed connection. The method that is used in this proof of concept is sending SQL queries directly to the database.

4.1.2 Client Database

The database system used on the Android-devices is SQLite. Due to the nature of Android and mobile devices, the creation and maintenance of the database have to be done in the code of the application, unlike a normal database server where the application and the database is two separate things. Although the architecture is different from the way of working with most other **DBMS**(Database Management System), the general principals are the same.

When the database is first used, the **onCreate()** method is called which creates all the tables, triggers, etc. Afterwards, the database is running on the device and if changes are necessary, they have to be implemented using the **onUpdate()** method. This method is invoked by increasing the version number of the database, so when the application is running it checks it's own version with the version of the database on the device. If differences exists, it will run the changes on the schema, and these changes are made using the normal statements for SQL.

```
@Override
public void onCreate(SQLiteDatabase sqLiteDatabase) {
String createUserTable = "CREATE TABLE User (\n" +
        "  id INTEGER PRIMARY KEY,\n" +
        "  person INT NOT NULL,\n" +
        "  password CHAR(45) DEFAULT NULL,\n" +
        "  FOREIGN KEY(person) REFERENCES Person(id));";

    sqLiteDatabase.execSQL(createPersonTable);
        }

@Override
public void onUpgrade(SQLiteDatabase sqLiteDatabase, int i, int i2) {
    Log.w(DBHelper.class.getName(), "Upgrading database from version " +
        i + " to "
            + i2 + ", which will destroy all old data");
}
```

List 4.3: SQLite create and update method

When the database is running, the application uses it by requesting a database-connection from the SQLiteOpenHelper, which it can then use to read/write data to the database. Once the connection is no longer needed, it will be closed with a close-command.

4.1.3 Synchronization

The synchronization process implemented in the prototype is only one-way, from the database to the Android application. The process is separated into 3 parts; the database connection, as explained in section 4.1.1; the SQL query; and call for inserting data in the local database. The last two parts will be explained further in this section.

As mentioned, the SQL queries are sent to the database directly. The executing of SQL queries on the database is a two step process, first a statement is prepared, which is a text string containing the entire SQL query. Then, the query is executed using one of the following methods on the connection, **executeQuery()**, **executeUpdate()** or **execute()**. **executeQuery()** is used when a select query is called and a result set is expected in return. **executeUpdate()** is used when running an update, which returns the number of rows affected. Lastly, **execute()** is used when the type of query is dynamic and therefore unknown at compile time. It returns either true or false, depending on whether it is a select or update/delete/insert query, respectively. Common for all these is that they need to be passed a string containing the SQL query to run. An example of the creation and execution of a statement can be seen in listing 4.4

```
        String lastSync = db.getLastSync();
        String sql = "SELECT * FROM Person " +
                    "WHERE timestamp > '" + lastSync +"'";

        ResultSet rs = st.executeQuery(sql);
```

List 4.4: creation and execution of a statement

The **getLastSync()** returns the time of when it last successfully synchronized with the database server. This information is stored in a single row in the client database and is updated upon completion of synchronization. The resultset that is returned from the executeQuery-method contains all the rows of the result from the SQL query. In case of the example in listing 4.4, it would contain all the rows from person that have been altered in some way since lastSync. To access the data, a while loop is used to iterate through all the rows in the resultset, as shown in listing 4.5. The getInt(n)- and getString(n)-method returns the content of the column with index n as either an int or string respectively. In the example, a new row is inserted into the local database using the db.createPerson-method whose purpose and implementation is shown in listing 4.6. While the SQLite library does have methods for inserting and updating data, with the requirements for this prototype, it is easier to simple make a raw SQL statement and run it on the server directly.

```
            while (rs.next()){
                db.createPerson(rs.getInt(1), // ID
                                rs.getString(2), // name
                                rs.getString(3), // address
                                rs.getString(4)); //timestamp
            }
```

List 4.5: Iterating through the result set returned from the query

```
    public void createPerson(int id, String name, String address, String
        timestamp){
        String sql = "INSERT OR REPLACE INTO Person (id, name, address,
            timestamp)" +
                "VALUES (" + "'" + id + "', '" + name + "', '" + address +
                    "', '" + timestamp + "')";

        sqLiteDatabase.execSQL(sql);

    }
```

List 4.6: Inserting the new data into the database

A complete example of these methods can be found in apendix A.2.

5 Summary

5.1 Conclusion

During the first sprint, a working database, based on the wasteland project's schema [1], was implemented on the server. Furthermore a connection prototype was created to learn how to connect to the server database from an android tablet. To give the other groups the ability to share data, creation of a prototype to replicate the content of the server database to the client database was started. Several existing synchronization systems was researched to better understand how to implement synchronization for the clients. A functioning synchronization between the client's database and the server database has been confirmed possible. Although more time spent on implementing a working prototype would have been useful, the basic idea for constructing a complete synchronization system, adhering to the client's needs, was documented in chapter 3 of Sprint 1.

In conclusion, the sprint's different goals, as described in section 1.3 of Sprint 1, was achieved.

5.2 Reflection on Development Method

As the sprint was very short and we predicted the tasks to mostly include debugging, we concluded that they would not be a good indicator for measuring our *velocity* for future sprints. Therefore we chose not to use any of the development methods from the previous semesters.

The purpose of the first sprint was simply to make the existing code 'work'. Since no database existed on the server and the existing code library facilitating the communication between the tablets and the database, had critical design errors, we chose to redesign it.

As we had no way of estimating the tasks needed, we ended up with a lot of unfinished tasks at the end of the sprint. Our solution for the library communicating with the database were only at the stage of a working prototype, and we had not had any time to write on the sprint chapters in the report. As a consequence, the rest of the work had to be postponed to the next sprint.

In future sprints we plan to use scrum with some of the practices from Extreme Programming, like user stories and planning poker, to better estimate our work.

Part III

Second Sprint

1 Introduction

The purpose of this section is to give an overview of the chosen project for the current sprint. A description of the project's functionality and state will lead to a set of possible problems for the group to solve. These problems will be explained and lastly, a reasonable and feasible subset of problems will be chosen as the focus of the sprint.

1.1 State of Affairs

For the second sprint, the group chose to continue working on the synchronization problem. The project at this time contains a working MySQL database for the server and a working SQLite database for each client. These database schema are identical. Additionally, a description of how to implement a synchronization system and a prototype of this system exists.

1.2 Current Problems

As the backlog got cleared in the previous sprint, the tasks at hand is the functions needed in the different steps of the designed synchronization process, which are:

- Implement the different functions needed in the synchronization library.
  - Synchronize()
  - LastUpdated()
  - DownloadUpdatesFromServer()
  - UploadUpdatesToClient()
  - DownloadUpdatesFromClient()
  - UploadUpdatesToServer()
  - UpdateLastSynchronization()
- Implement a dummy-application to initialize the synchronization process in the library.

The functions mentioned above, and as introduced in section 3.2 of Sprint 1, will be fully described in the design chapter once they are chosen for implementation.

1.3 Sprint Goal

The primary goal of this sprint, is to accomplish one-way synchronization, meaning that clients can request and save data from the server, but are unable to upload any data. A secondary objective is to implement a dummy-application from where we can initialize the synchronization process. In relation to the sequence-diagram presented in Sprint 1, these goals are step 1 through 5 and step 10. Furthermore we have discovered documentation for a different synchronization protocol called *Synchronization algorithms based on message digest* (SAMD) [9]. We intend to investigate this to determine if it could be a superior alternative to the currently designed protocol. Thus the goals for sprint 2 are:

- Implement one-way synchronization
- Research SAMD
- Implement a dummy-application to initialize the synchronization process

2 Analysis

2.1 SAMD Algorithm for Database Synchronization

In continuation of the analysis done in sprint 1, research on solutions for a synchronization system will extend into this chapter.

SAMD [9] works by creating a companion table for each datatable that needs to be synchronized.

A companion table contain a hash-value for each row in the datatable. These values are generated from data in the corresponding row and stored on the server, to be compared to a new hash-value sent by a client. If a row's value, sent by the client, doesn't match the server's, the client is in need of synchronization.

2.1.1 Analysis of SAMD

Compared to similar algorithms that are commercially available, the SAMD algorithm is faster by an average of 0.64 seconds. [9, p. 397] Additionally the algorithm does not require that the two databases that are being synchronized are running the same database management system whereas many of the commercially available solutions such as Oracle do. Also, SAMD does not add extra columns to the datatables, instead it stores this data separately in the companion tables. Although this requires more storage space than some of the other solutions, storage space can for the most part be considered relatively inexpensive.

2.1.2 Database Structure

All information needed to implement SAMD will be stored on the server, and no structural changes are therefore necessary on the clients. The structural changes needed on the server to support SAMD will now be described. A set of companion tables are created for each client, where the set contains a companion table for each data table, e.g. the *Pictogram*-table, on the client. Every companion table for a client (CT_{Client}) contains a ***Flag*** representing whether a row contains changed data and a ***Hash*** value for each row in it's corresponding data table (DT_{Client}), as show in table 2.1. And for every part of the data tables primary key, composite or not, an attribute representing that part is added to the companion table. That is, $PK_1, PK_{...}, PK_n$ is the primary key of DT_{Client}, and ***type of*** $PK_1, PK_{...}, PK_n$ is the data type used to store each part of the primary key.

Likewise, a set of companion tables are created for the server, where the set contains a companion table (CT_{Server}) for each data table on the server (DT_{Server}), as shown in table 2.2. An additional attribute is needed to represent the client whom this row belongs to. This is necessary to keep track of the last state of a DT_{Client}, which we need to ensure a correct analysis of changes since last synchronization.

Attributename	**Type**	**Nullable**
$\underline{PK_1}$	type of PK_1	false
$\underline{PK_{...}}$	type of $PK_{...}$	false
$\underline{PK_n}$	type of PK_n	false
hash	char(32)	true
flag	tinyint	false

Table 2.1: CT_{Client} - PK$_i$ is a prime attribute of the primary key from DT_{Client}

Attributename	**Type**	**Nullable**
$\underline{PK_1}$	type of PK_1	false
$\underline{PK_{...}}$	type of $PK_{...}$	false
$\underline{PK_n}$	type of PK_n	false
hash	char(32)	true
flag	tinyint	false
<u>ClientID</u>	integer	false

Table 2.2: CT_{Server} - PK$_i$ is a prime attribute of the primary key from DT_{Server}

2.1.3 Comparing Rows using MD5

The hash-function we use, to determine where changes are made, is MD5, which is a one-way hash function belonging to a group called message digest algorithms. Using this algorithm on a row of data, we obtain a fixed length value that has a high probability of uniquely identifying the state of the row [10].

$$MD5("") = d41d8cd98f00b204e9800998ecf8427e \tag{2.1}$$
$$MD5("abc") = 900150983cd24fb0d6963f7d28e17f72 \tag{2.2}$$

Once a hash value has been generated for a row, it is saved in the companion table, as illustrated in table 2.3 for clients and in table 2.4 for the server. By comparing the hash value generated for a row in DT_{Client} on synchronization, with the hash stored in CT_{Client}, we can determine which rows has changed on the client since last synchronization. The same can be applied for DT_{Server} with CT_{Server} to determine the changes on the server.

DT_{Client}

PK	$Column_1$	...	$Column_n$
1	Tom	.	400
2	John	.	200
.	.	.	.
.	.	.	.
.	.	.	.
m	.	.	.

CT_{Client}

PK	**Hash**	**Flag**
1	j345jh43kh53	0
2	fdg6fg90sfhk	0
.	.	.
.	.	.
.	.	.
m	.	.

Table 2.3: A data table and it's corresponding companion table for a client

2.1.4 The Algorithm

To Synchronize two datatables, DT_{Client} and DT_{Server}, the SAMD algorithm requires a companion table for each, called CT_{Client} and CT_{Server} as mentioned earlier.

DT_{Server}

PK	$Column_1$	...	$Column_n$
1	Tom	.	400
2	John	.	200
.	.	.	.
.	.	.	.
.	.	.	.
m	.	.	.

CT_{Server}

PK	**Hash**	**Flag**	**ClientID**
1	j345jh43kh53	0	42
2	fdg6fg90sfhk	0	42
.	.	.	.
m	.	.	.
1	j345jh43kh53	0	43
2	fdg6fg90sfhk	0	43
.	.	.	.
m	.	.	.

Table 2.4: A data table and it's corresponding companion table for the server

The algorithm performs three steps to complete synchronization:

- first, DT_{Client} is synchronized with CT_{Client}
- then, DT_{Server} is synchronized with CT_{Server}
- and finally, the two companion tables are examined to determine which rows in the two datatables need to be updated and how.

Step one: Synchronize DT_{Client} with CT_{Client}

The first synchronization step, as illustrated in figure 2.1, is between the clients datatable and its companion table located in the server database. For each row of the datatable the client calculates a hash-value and sends it to the server database. The server database compares the hash-value to the value in CT_{Client}, if the two are different the flag column is set to 1 and the new hash-value is stored in the hash-column. If there exist a row in CT_{Client} with a key that does not appear in DT_{Client}, the hash-column is set to *Null* and the flag is set to 1, indicating that the row was deleted.

DT_{Client}
DT_{Server}
CT_{Client}
CT_{Server}

Figure 2.1: SAMD: Step one, synchronize DT_{Client} with CT_{Client}

Step two: Synchronize DT_{Server} with CT_{Server}

The second synchronization step, as illustrated in figure 2.2, is between the datatable on the server database and its companion table. As the companion table contains values for all clients, only the rows with a *ClientID*-value corresponding to the client is synchronized with the datatable. As before, for each row in DT_{Server} a hash-value is calculated and compared to the value in CT_{Server}, and if the two are different, the hash-value is updated and the flag is set to 1 in CT_{Server}. If a row in DT_{Server} is deleted the hash-value in CT_{Server} is set to null and the flag is set to 1.

DT_{Client}
DT_{Server}
CT_{Client}
CT_{Server}

Figure 2.2: SAMD: Step two, synchronize DT_{Server} with CT_{Server}

Step three: Synchronize DT_{Server} with DT_{Client}

In the final synchronization step, as illustrated in figure 2.3, the two datatables are synchronized using the information gained from the two companion tables. The algorithm performs a full outer join on CT_{Server} and CT_{Client} tables with the condition that the flag-attribute must be set to 1, in at least one of them.

Each row in the datatables is treated differently depending on where it has been changed since the last synchronization. If only the row in DT_{Server} has been changed, the row on DT_{Client} will be replaced with the one on DT_{Server} and the flag in CT_{Server} will be set to 0, indicating that the two rows are now synchronized. The hash-value from CT_{Server} is also copied into the CT_{Client} hash-value. If the row in DT_{Server} has been deleted (i.e. the hash-value in CT_{Server} is *Null*) the corresponding row in DT_{Client} is deleted. Likewise, the rows in the two companion tables representing the deleted datatable-row are deleted. If only the row in DT_{Client} has been changed it is synchronized in a similar fashion. However, when both flags in CT_{Server} and CT_{Client} has been set to 1, we have a merge conflict. The algorithm uses a *synchronization policy* to determine which row takes precedence. These policies will be described further in chapter 3.1.1 of sprint 3.

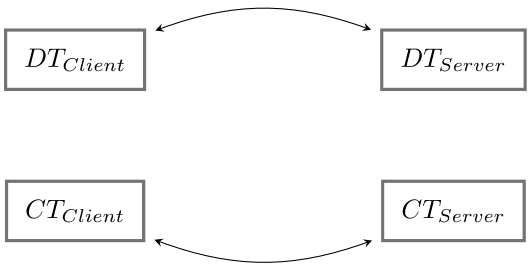


Figure 2.3: SAMD: Step three, synchronize DT_{Server} with DT_{Client} based on information stored in CT_{Server} and CT_{Client}

3 Design

To help spread data among the different groups in the multi-project environment, a temporary solution for downloading content from the server to the clients was needed. The solution would only have to support one-way synchronization and this will be designed in the following sections and implemented later in the sprint. Based on the idea of a complete timestamp-synchronization process, as described in 3.2, the first five steps handling one-way data-synchronization, from server to client, will be implemented. As several approaches are now being considered, the timestamp-solution will now be referred to as *TSync*.

3.1 Designing Timestamp Synchronization - Part 1:2

This section will describe the design of step 1 through 5 and step 10, as illustrated in the sequence diagram 3.3 of sprint 1.

3.1.1 The Synchronization Function

The synchronization library will contain a single function accessible from outside the library called Syncronize(). The purpose of this function, is to initialize and structure the synchronization process. This function will also need to save $Sync_{Time}$ temporarily for reference, containing the exact time of when this function was called, and the reason for this will be explained further in section 3.1.5. The synchronization process is divided into steps and therefore needs to call several other functions to complete it's purpose. An overview of these functions was given in section 3.2 of the first sprint.

3.1.2 The LastUpdated Function

The purpose of this function is to return a timestamp which represents when the client database was last updated. This value, referenced to as $PrevSync_{Time}$, will be stored in the clients database. As there currently are no need to store each specific timestamp whenever a synchronization have completed, no new record will be inserted on these events. Instead, the record with id 1 will be updated to contain the latest timestamp. The table-design for containing this value is illustrated in table 3.1. This table have no relations to any other table in the database.

Attribute name	**Type**	**Nullable**
id	integer	false
timestamp	timestamp	false

Table 3.1: The table *Synctable* representing time of last synchronization.

3.1.3 The DownloadUpdatesFromServer Function

This function will request new content by first establishing a connection to the server database and running a SQL-command to fetch $LastChange_{Time}$, which is a timestamp representing when a change was last performed in the server database. If $PrevSync_{Time}$ is equal to or later than $LastChange_{Time}$ then there are nothing new to fetch and it skips to the SaveSynchronization-Timestamp function. Otherwise any records containing a timestamp later than $PrevSync_{Time}$ will be fetched. These sets of data will be stored momentarily in the synchronization library, referenced to as $Data_{Server}$, for use in the next stage of the synchronization process.

3.1.4 The UploadUpdatesToClient Function

The UploadUpdatesToClient-function, as the name suggests, takes the data from $Data_{Server}$ and tries to save it to the client's database. This is done by establishing a connection to the client's database and using several SQL-statements to insert and update data in their respective tables. If this operation does not succeed, then, depending on the cause, the operation will be run again or the synchronization will be aborted.

3.1.5 The SaveSynchronizationTimestamp Function

This function will be called as the very last step in the synchronization process and it's only purpose is to save the exact time when synchronization took place, known as $Sync_{Time}$. It's worth considering when this time actually is, as it could be at several different stages, e.g. at the moment synchronization is initialized, at the moment updates have been received from the server or once all steps have been completed. One scenario which could become an issue, is that large amounts of data will at times be needed to be sent back and forth during a synchronization. This in itself is not a problem per se, but a user on the client could potentially create additional data while synchronization takes place in the background. To make sure this new data will be synchronized at a later time, $Sync_{Time}$ must not be set during or after synchronization.

Therefore, the best time to set $Sync_{Time}$ is when the function Synchronize() is called from an external source. All content created before this time will be fetched from the server and any new content added to the server during or after synchronization will be downloaded at a later time. $Sync_{Time}$ will be saved to the client's database, to the table called *Synctable*.

4 Implementation

4.1 Implementing Timestamp Synchronization - Part 1:2

This section describes parts of the implementation previously designed.

4.1.1 The Synchronization Function

To allow the user to operate the device during synchronization, it runs as a background service, and is completely invisible to the user. The synchronization method is called to start the synchronization service, which then takes care of downloads and insertions in the client database. The service runs the download for each table as separate Android AsyncTasks [11]. These tasks are scheduled to be run in serial on a single thread, which allows each table to be queued so that they are downloaded in the right order to avoid foreign key constraints. These different parts of the synchronization process will be explained in the following sections.

4.1.2 The DownloadUpdatesFromServer Function

Downloading the data from the server to the client is a three step process. This splitting of the download-process is due to memory limitations posed by working on the Android platform, which will be discussed in greater detail, in section 3.1.3 of the third sprint. To avoid unnecessary traffic between the server and the client, the client asks the server if there are any changes since the last time it connected. The client calls a stored procedure on the server using $PrevSync_{Time}$ as input, and the server returns 0 or 1 based on whether there have been any changes since last synchronization. If changes exists, the client requests the server for the IDs of all the rows containing changes. This list of IDs is then broken into batches for download according to the size limitations posed by the memory constraints.

As can be seen in listing 4.1, there are four different cases for getting the IDs of the rows that contain changes. To minimize traffic, only the ID of the rows are downloaded at first. Most tables follow a convention for naming primary keys, which allows for dynamic generation of select statements. However, there are a few tables that do not adhere to this convention, and therefore need special case to make them work. It might have been smarter to rename all the tables to follow the same convention, but this would involve changes that would impact a lot of other groups, resulting in refactoring work for these. Thus, the workaround was deemed a better solution for the multi-project.

```
if(params[0].equals(GuardianOfTable.TABLE_NAME)){
        sql = " SELECT " + GuardianOfTable.COLUMN_IDGUARDIAN + ", "
        + GuardianOfTable.COLUMN_IDCHILD + " FROM " +
        params[0].toString() + " " + "WHERE timestamp > '" +
        params[1].toString() + "'";
}else if(params[0].equals(AdminOfTable.TABLE_NAME)){
        sql = "SELECT " + AdminOfTable.COLUMN_IDUSER + ", " +
        AdminOfTable.COLUMN_IDDEPARTMENT +       " FROM " +
        params[0].toString() + " " + "WHERE timestamp > '" +
        params[1].toString() + "'";
}else if(params[0].contains("_")){
        String id1 = params[0].substring(0,params[0].indexOf("_")) + "_id";
        String id2 = params[0].substring(params[0].indexOf("_") + 1) + "_id";

        sql = "SELECT " + id1 + ", " + id2 + " FROM " + params[0].toString()
         + " " + "WHERE timestamp > '" + params[1].toString() + "' ";
}else{
        sql = "SELECT " + params[0].toString() + "_id FROM " +
        params[0].toString() + " " +
        "WHERE timestamp > '" + params[1].toString() + "' ";
}
```

List 4.1: The four cases to get a list of IDs, from the server, that contain changes.

The next step is to create the SQL queries based on the returned list of IDs, and download the rows that have been changed. The example code in listing4.2 shows how the queries for downloading from tables with composite primary keys. The code for single primary keys are very similar but with fewer variables. It breaks the available IDs into batches and creates a query for each batch. As the IDs are added to the query they are removed from the list of available IDs. Once done, the queries are executed one at a time and the results are inserted into the client database as described in section 4.1.3

```
while(updates.size() > 0){
        int bounds;
        sql = " SELECT * FROM " + tableName + " " + "WHERE ";
        if(updates.size() > downloadLimit){
                bounds = downloadLimit;
        }else{
                bounds = updates.size();
        }
        for(int i = 0; i < bounds; i++){
                if(i ==0){
                        sql += "" + rsmd.getColumnName(1).toString() +
                        " = " + updates.get(i)[0];
                        sql += " AND " + rsmd.getColumnName(2).toString() +
                        " = " + updates.get(i)[1];
                }else{
                        sql += " OR " + rsmd.getColumnName(1).toString() +
                        " = " + updates.get(i)[0];
                        sql += " AND " + rsmd.getColumnName(2).toString() +
                        " = " + updates.get(i)[1];
                }
                remove.add(updates.get(i));
        }
        sql += ";";
        result.add(sql);
        updates.removeAll(remove);
        remove.clear();
}
```

List 4.2: An example showing the creation of the queries for downloading the changed data.

4.1.3 The UploadUpdatesToClient Function

Because of the difference between MySQL and SQLite, and the way the data in the result set is handled, it is not possible to just insert the data into the client database, and a few special case are needed to handle the exceptions. The data is inserted using prepared statements, which can be described as a template for a specific SQL query, and is effective when the query needs to be run a lot time with different data. A prepared statement have the syntax shown in listings 4.3, where the question marks will be replaced by the data. This approach also have the added advantages of reducing the risk of SQL injection. While this is not something that one would expect when looking at how the application is going to be used, it is a nice side effect.

```
INSERT OR REPLACE INTO 'tableName' VALUES (?,?,?)
```

List 4.3: Prepared statement example

The creation of the prepared statement is quite simple as seen in listing 4.4, after this the variables are bound to the statement with some special cases based on the type of data. These different cases are shown in listing 4.5. The first case is null values. Since a result set returns text it is necessary to specifically insert null. It is the same problem with blob values. If the blobs where just inserted as part of a normal insert query, it would be converted to text, which would compromise the data. Therefore, the blob is converted to byte data before being bound to the statement.

```
String sql = "INSERT OR REPLACE INTO " + rsmd.getTableName(1) + " ";
sql += " VALUES (";
for(i = 0; i < rsmd.getColumnCount(); i++){
    if(i > 0)
    {
        sql += ",";
    }
    sql += "?";
}
sql += ")";

SQLiteStatement statement = db.compileStatement(sql);
statement.clearBindings();
```

List 4.4: Creation of the prepared statement

The last special case is needed as SQLite does not support enums. The role of a profile is stored as an enum, and the only way to get the numeric value of an enum from MySQL is to specifically ask for it. So we had to either make a special SQL query for requesting the data from the tables with enum, or handle it when it is inserted into the client database. As can be seen from the code in listing 4.5, it was the latter that was chosen. While neither of the approaches can be considered optimal, making the changes to the data structure was deemed to influence too many other projects, and therefore this solution was chosen.

```
for (i = 1; i < rsmd.getColumnCount() + 1; i++){
    if(rs.getString(i) == null){
        statement.bindNull(i);
    }else if(rsmd.getColumnType(i) == Types.LONGVARBINARY){
        Blob blob = (Blob)rs.getBlob(i);
        int blobLength = (int)blob.length();
        byte[] blobByte = blob.getBytes(1, blobLength);
        statement.bindBlob(i, blobByte);
        blob.free();
    }else{
        if(rsmd.getColumnName(i).equals("profile_role")){
            int role = 3;
            if (rs.getString(i).equals("Admin")){role = 0;}
            else if (rs.getString(i).equals("Parent")){role = 1;}
            else if (rs.getString(i).equals("Guardian")){role = 2;}
            else if (rs.getString(i).equals("Child")){role = 3;}
            statement.bindString(i,Integer.toString(role));
        }else{
            statement.bindString(i, rs.getString(i));
        }
    }
}
```

List 4.5: Special cases regarding insertion

5 Summary

5.1 Conclusion

In this sprint we have successfully achieved our three goals. The first goal of implementing the one-way synchronization has been described in section 4. For our second goal we have successfully researched and described SAMD in section 2.1, which claims to be a superior solution. Our third goal, implementing a dummy-application, was completed but it is only used for testing purposes, and other means will be given to groups in need of initializing the synchronization process.

5.2 Reflection on development method

During this sprint we used planning poker to estimate our pending work-assignments. This turned out to take longer than we expected, accumulating to about 20 percent of our available project-time. Furthermore, the relatively short time of the project, combined with our level of experience, meant we didn't gain anything by estimating. As a result, we found it wasteful to continue using this practice and agreed to completely avoid estimating in future sprints. We will, however, continue to manage work-assignments to give a clear overview of what assignments still needs completion, and whether someone is working on it. Likewise, we will continue to participate in the weekly SCRUM meeting with the other multi-project groups. Although we mostly skipped stand-up meetings this sprint, the supposed benefits gained from this practice is considered a fair use of our time and thus we intend to keep it.

Part IV

Third Sprint

1 Introduction

The purpose of this section is to give an overview of the chosen project for the current sprint. A description of the project's functionality and state will lead to a set of possible problems for the group to solve. These problems will be explained and lastly, a reasonable and feasible subset of problems will be chosen as the focus of the sprint.

1.1 State of Affairs

For the third sprint the group chose to continue working on the synchronization problem. The project at this time consists of a working MySQL database on the server, a working SQLite database for each client and an android service allowing the clients to update themselves to fit changes in the server database. The schema has been adapted with timestamps to facilitate the current one-way synchronization protocol used. Furthermore, research into different synchronization approaches have been described and concluded.

1.2 Current Problems

For this sprint the primary problem is to determine which synchronization approach to use, and implement it. The two candidates are: our currently used timestamp-based approach (TSync) and the SAMD approach. Another problem is to integrate our synchronization with the other groups' projects. We also have yet to implement an ID-handling method, and find a method for dealing with merge-conflicts. This leaves us with the following tasks:

- Decide on the synchronization approach
- Implement the chosen synchronization approach
- Implement ID-handling
- Determine how to deal with merge-conflicts

1.3 Sprint Goal

At the beginning of this sprint we have two synchronization approaches under consideration. The timestamp-based solution we designed, and the SAMD design. We need to decide on what to do in the case of merge-conflicts. This makes the goals for this sprint the following:

- Decide on strategy for handling merge-conflicts and implement it.
- Determine criteria for comparing the different synchronization approaches and chose one to implement.
- Implement the chosen synchronization approach.

2 Analysis

According to the research on SAMD, as mentioned in section 2.1.1 of the second sprint, their solution is faster than other available algorithms. But, after having studied the details of SAMD, we question this statement. SAMD appears to store and send a lot of additional information from the client to the server. We will therefore calculate this *overhead* for each approach and compare the results.

The goal of this comparison is to find the best solution for the customer.

2.1 Transmission Overhead

The two approaches have different transmission overhead, by this we mean the amount of data being transmitted in excess of the changed rows.

When using the TSync approach, the client only needs to send a single timestamp to the server to start the synchronization. If changes exists on the server, all IDs from these rows are downloaded to the client, as described in 4.1.2 of the second sprint. While each row contains its own timestamp, the timestamp is unique to each device and should therefore be omitted from the transmission As a result, the total overhead is a single timestamp, using 4 bytes.

When using SAMD, all rows on a client is sent to the server, but no additional information about the rows are transmitted.

The resulting overhead can be viewed in table 2.1, where

- n is the total size of the database
- m is the total size of changed rows
- x is the sum of storage needed for storing the set of attributes in PK
- y is the total number of changed rows.

	Overhead
TSync	4 bytes $+ x \times y$
SAMD	$n - m$

Table 2.1: Test results for **Transmission overhead** when using TSync and SAMD

2.2 Structural Overhead

Structural overhead is the size of the stored information needed solely for synchronization purposes.

To calculate the structural overhead added by the two approaches, the different data types of each needed attribute will be considered.

TSync requires that all entries of every data table contains a timestamp. Additionally a timestamp is saved on each client to keep track of their last successful synchronization attempt. A timestamp takes up 4 bytes of storage in MySQL [12]. In SQLite, the timestamp is represented as the datatype *Integer* which takes up 4 bytes of storage [13].

Let n be the amount of rows in the set of data tables and notice that the server and each client have the same set of data tables. The resulting structural overhead can be viewed in table 2.2.

	Overhead
On Server	$n \times 4$ bytes
On each Client	$n \times 4$ bytes $+$ 4 bytes

Table 2.2: **Structural overhead** as a result of using TSync

SAMD does not store any information regarding synchronization on the clients, therefore we only need to consider the server in this case, which is running MySQL. On the server, no attributes are added to any data tables, but a companion table for each data table, and for each client, is created. This companion table consists of a set of attributes (PK) to construct the primary key copied from it's corresponding data table and two additional attributes, ***Hash*** and ***Flag***. The data types of PK is copied from the corresponding data tables, so no direct value, for storage usage, can be given here. Thus we denote this cost as x. The hash-value is saved as a char(32), taking up 16 bytes of storage [12]. The flag-attribute is stored as either 0 or 1, taking up 1 byte of storage [12]. Therefore, every row of data on a client, takes up $x + 17$ bytes on the server. The companion table for the servers data tables contains an additional attribute, the *ClientID*, which is an *Integer* representing a client. An integer costs 4 bytes [12]. As a result, this companion table takes up $x + 21$ bytes of data, for each existing row per client.

The resulting structural overhead can be viewed in figure 2.3, where

- n is the set of rows from all data tables on a client.
- m is the set of rows from all data tables on the server.
- c is the amount of clients
- x is the sum of storage needed for storing the set of attributes in PK

	Overhead
On Server	$\sum_{i=1}^{c}(n_i \times (x + 17 \text{ bytes})) + m \times (x + 21 \text{ bytes})$
On each Client	0 bytes

Table 2.3: **Structural overhead** as a result of using SAMD

2.3 Deciding on a Synchronization Approach

By looking at the transmission-overhead, we see that our suspicions were confirmed, that SAMD indeed does send a lot of additional data. Likewise, the structural-overhead of SAMD is much greater compared to TSync. It is likely that the structural-overhead of SAMD is part of what makes it faster to execute, but clearly, the transmission-overhead will make it slower over a WIFI connection. Whether the speed is noticeably slower is dependent on several factors, such as the speed of the WIFI connection and the data needed for synchronization.

However, these limitations of SAMD seems to be easily avoided, if we decide to generate the hash-values locally before sending them to the server. Although we have confirmed our suspicions, we are unable to conclude whether an optimized version of SAMD is a better or worse solution for the customer.

To determine which one is the best solution, we want to implement a functional prototype for both approaches, and once completed, perform tests on these prototypes to measure which one is best for the clients purposes. These tests will be performed in the following sprint, if possible.

To help decide which approach to use, we will analyze the most important criteria to test for.

Therefore, we will now set up some criteria for comparing the two approaches, TSync and SAMD, to be implemented in an WIFI-environment.

2.4 Criteria

To find proper criteria for comparing, it's important to first look at the environment where the subjects SAMD and TSync will be used. As mentioned earlier, the synchronization process will run on Android devices and take place over a WIFI connection. Therefore, one criteria to test for is *time*. The less time it takes to complete an attempt, the lower the chances of getting the connection interrupted, and the lower the risk of problems with concurrent access. Another criteria is *memory usage* during synchronization. High memory usage increases the risk of crashing the process by running out of memory on the device.

The criteria for testing is summarized as:

- Time
- Memory Usage

To see how each candidate works in different situations, the parameters of the tests will differ in:

- Speed of the connection
- Size of data
- Number of relations
- Memory available on device
- Processing power available on device

The candidates will, however, always be compared on tests performed under identical conditions.

A table for each criteria will be filled out for every candidate, so a comparison can be performed and a solution chosen.

3 Design

3.1 Conflict And Error Handling

This section contains a description of synchronization conflicts and errors and how these can be handled.

3.1.1 Merge Conflicts

A merge conflict occurs when a client receives data from the server, which is older than what is located in the client's database. Likewise, the server can receive data from a client, which is older than what is located in the server's database. This conflict happens when two or more clients alters the same data and then tries to submit the changes to the server. The first client submitting will not get into a merge conflict as no changes have been made on the server yet, but all following client's will get the merge conflict. The merge conflict is illustrated in table 3.1 where three clients A, B and C all tries to change the data element $Data_B$, resulting in a merge conflict for *Client A* and *Client C* as *Client B* was first to commit its changes.

	Client A	**Client B**	**Client C**
1	Synchronize()		
2	Success		
3		Synchronize()	
4		Success	
5			Synchronize()
6			Success
7			Change $Data_A$
8			Synchronize()
9			Success
10			Change $Data_B$
11		Change $Data_B$	
12		Synchronize()	
13		Success	
14	Change $Data_B$		
15	Synchronize()		
16	Failure, merge conflict		
17			Synchronize()
18			Failure, merge conflict

Table 3.1: The merge conflict illustrated

To resolve the conflict, a decision must be made so the synchronization process can continue and complete. The decision can be as simple as keeping whatever data already exists in the database, or overwriting the existing data with the conflicting data.

Another option would be to let the server make an educated guess as to what data to keep and what to discard. To help the server make the choice, additional information about data-changes could be stored, e.g. if the server saves a timestamp for when the clients made the changes locally, it could compare the two and let whoever made changes last get to win the merge conflict and keep the data. For example, using the example in table 3.1, even though *Client C* synchronizes last (at 17), the changes made to $Data_B$ (at 10) is older than the changes made by *Client A* (at 14), and it is therefore likely that the server wants to keep the changes made by *Client A*. This decision can become very complex when taking several factors into account, and is at the moment of low priority. For now, the synchronization process will use a simple policy to handle merge conflicts, while reserving the possibility of changing it later if time permits.

The policy chosen, is to always overwrite data when a merge conflict is encountered. Thus the data-changes of the client that synchronizes last is kept.

3.1.2 Disruptions

A problem that needs to be handled is loss of connection. As the clients will be synchronizing over WIFI, it is likely that the connection at times will be interrupted or completely dropped. This is likely, as the clients will be used in both in- and outdoor situations, and will at times leave the institution entirely. If a synchronization is in progress while the connection is lost, one of two things should happen. Either, the entire synchronization process should be rolled back and thereby restoring the database to its former state, or, the system should save the progress and try to resume at a later time. At first glance, the second option seems more appealing as the time needed for finalizing the synchronizing would be reduced. But, several new potential problems emerges, as some data may contain relations to other data which was not synchronized. Furthermore, synchronizations are not expected to contain large amounts of data, but rather small data-updates received daily, with the exception of first-time synchronizers, who will have to receive the entire database content before being used. Therefore, the solution to handle connection disruptions will be to wrap the synchronization process in a transaction [14], so it may be rolled back if needed. This will also be our default solution to handle any other errors.

3.1.3 Memory Constraints

Most Android devices are restricted in regards to memory, at least when compared to desktop computers. Therefore, as a standard, applications are restricted to the amount of memory they are granted, and because there are many different Android devices with widely different specifications, it is not possible to know beforehand how much memory your application will be allocated. It is therefore necessary to takes this memory management into consideration during development. Android can manage this, up to a certain point, by killing processes that are no longer used. However, it is better to handle this problems within the application itself, because Android does not discriminate between different processes. Thus, it might kill a process that the application needs and, as a consequence, hurt the user experience or even make the application crash. [?]

The memory constraint becomes apparent when there are large updates on the server e.g. when the application is first connected to the server. The amount of data needed to be downloaded can be several hundreds of megabytes, and trying to download this in a single query could exceed the memory allocated to the application and make it crash. There is a way to allocate more memory to an application [15], but since the total available memory is still uncertain, it is not a proper solution. Android encourages developers to stay away from this feature unless there are no other options. The way the problem was solved in the client, was to split the downloads into smaller batches, as explained in section 4.1.2 of the second sprint.

To determine the size of these batches a series of tests where conducted to measure both the memory usage on the device and the total run time for the query. While increasing the number of queries to the database will increase the amount of traffic between the client and the server, this was deemed to be negligible(less then 1kb/s) and the total data to be downloaded was the same. The test was conducted by measuring the memory used by the application. Android SDK have a build-in class for getting information like memory(Debug.MemoryInfo) and while this is not the exact memory used by the application it is a close enough estimate to use in this test.

```
pids[0] = android.os.Process.myPid();
android.os.Debug.MemoryInfo[] memoryInfoArray =
    activityManager.getProcessMemoryInfo(pids);
Log.i(TAG, "Total Pss: " + memoryInfoArray[0].getTotalPss() + "\n");
```

List 3.1: Retriving memory usages for a given process

The method *activityManager.getProcessMemoryInfo(pids)*, as shown in listing 3.1, is passed an array of process IDs and returns the memoryInfo array which contain the information for the supplied IDs. The Pss(Proportional Set Size) is the amount of memory (kB) Android believes that the process is using [?]

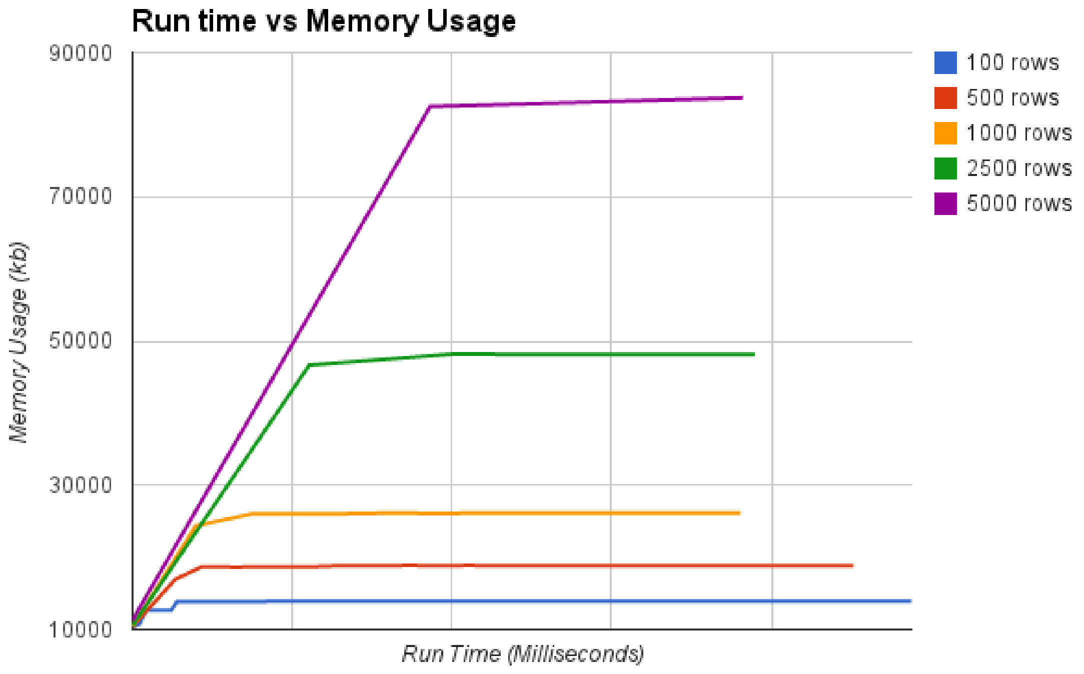


Figure 3.1: Comparisen of run time vs memory usage

When looking at the test results, shown in figure 3.1, there are some clear correlations between the size of the batches and the memory used by the application, but there are also a difference in the overall run-time and the size of the batches. A closer look at the results, show that there are no significant difference in the run-time of 1000, 2500 and 5000 rows, but the increase in memory

Rows	Runtime(ms)	Percentage decrease	Memory Usage(kB)	Percentage increase
100	73130	–	13935	–
500	67697	7.43%	18826	35.1%
1000	57089	21.93%	26114	87.4%
2500	58410	20.13%	48163	245.63%
5000	57346	21.58%	83738	500.93%

Table 3.2: Total run time and average memory usage

usage for 2500 and 5000 rows is large when compared to 1000 rows. While it is still almost a double in memory, the actual usage is still low enough that it considered a good trade-of. Based on these tests, a maximum batch size of 1000 rows is selected to be the optimal batch-size when synchronizing.

3.1.4 Adjusting Synchronization Processes

The policy used for merge conflicts is implemented by reordering the steps in the synchronization process. The clients data-changes are uploaded to the server before the servers data is downloaded, as depicted in the new sequence-diagram in appendix A.4.

3.2 Designing Timestamp Synchronization - Part 2:2

The following sections will contain a design of the remaining steps for implementing timestamp synchronization. The steps not already designed for the synchronization library are, the methods for *Downloading Updates From Client* and *Upload Updates To Server*

3.2.1 DownloadUpdatesFromClient()

The purpose of the DownloadUpdatesFromClient() function is to fetch the data which has been altered locally since last synchronization. To fetch this data it requests all entries in the CDB where the timestamp is newer than $PrevSync_{Time}$. Since all entries update their timestamp when altered this will give us all the entries that has been altered since last synchronization. This data will be referred to as $Data_{Client}$ in the future.

3.2.2 UploadUpdatesToServer()

The purpose of the UploadUpdatesToServer() function is to upload the $Data_{Client}$ received from DownloadUpdatesFromClient(), to the server. This will be accomplished in full, or not at all, by using insert-statements, for each entry in $Data_{Client}$, nested in one transaction.

3.3 Designing SAMD Synchronization

In the original algorithm, all data in a table is sent to the server which then computes the CT_{Client}. Because some of the tables in the multi-project database contain large binary files this is very inefficient, as concluded in section 2.3, therefore we alter the design to consist of the following steps:

3.3.1 Step 1: Computing CT_{Server}

The first step is executed as a stored procedure on the server database. In this step the database computes the MD5 hash-value of all rows in the data tables, DT_{Server}, and updates or inserts the value in the corresponding CT_{Server} if necessary. For those rows that has been deleted in DT_{Server}, the hash-value is set to *null* in the corresponding row of the CT_{Server}. Finally, the flag-value is set to 1 for the rows that have been altered.

3.3.2 Step 2: Computing CT_{Client}

The second step has similar goals as the first step: To synchronize the clients data tables, DT_{Client}, with the corresponding companion table, CT_{Client}, on the server. However, to minimize the amount of data we need to send between client and server, we design it so that each CT_{Client} is stored on the client. The computation of the hash-values and updating of CT_{Client} is performed locally, following the same pattern as in step 1. A copy is then sent to the server for temporary use in step 3.

3.3.3 Step 3: Computing Result Table

The goal of step three is to compare each set of companion tables, CT_{Client} and CT_{Server} to determine which actions the client must take. By performing a full outer join of two companion tables on their row id, it is possible to determine which action should be taken for each row based on the placement of null values in the columns and the value of the two flags.

All of this is again intended to be implemented as a stored procedure in the server database, thereby taking advantage of the database query optimizer for the joins.

3.3.4 Step 4: Executing Actions

In the final step, the result table from step 3 is sent to the client and all the actions computed in the previous step is carried out by the client. This is done by executing SQL statements on both databases to perform the computed actions. Once an update has been carried out, the flag-values are set to 0 and the hash-values are updated. If a hash contains the value null, the corresponding row in the companion tables is removed.

4 Implementation

4.1 Implementing Timestamp Synchronization - Part 2:2

Development on the second part of the synchronization process was cut short due to focus being needed elsewhere. Therefore, no real improvement was achieved on the implementation of the synchronization. As a consequence, this section will focus on some of the prototyping that where carried out in the beginning of the sprint. So while these implementations are not necessary the ones that will appear in the final version, they show the basic principals and functionality needed to achieve the desired solution.

4.1.1 Unique Device ID

To allow for each device to assign IDs to new rows without worrying about duplicate IDs being created on different clients, each client is assigned a client ID when first connected to the server. This ID is then used to prefix any new row IDs. This is achieved through the following steps. First, the server holds a table containing the highest ID created to date. Whenever a client connects for the first time, it is assigned the next ID in the sequence, as shown in listing 4.1.

```sql
INSERT INTO deviceId (id, deviceIds) VALUES ('1', '1')
        ON DUPLICATE KEY UPDATE deviceIds = deviceIds + 1;

SELECT deviceIds INTO deviceId FROM deviceId WHERE id = 1;
```

List 4.1: Stored proceure for assigning new client id.

Since SQLite does not allow for defining a custom sequence for ID-generation, it is necessary to create a table on the client to keep track of what IDs have been assigned to each table. By using a trigger on insert-statements, the client assigns a correct ID to the newly created rows and incrementing the count for the table in question, see listing 4.2. This trigger increments the count for the Person table and assigns this new ID, with the client prefix to the new row.

```java
private static final String createInserTrigger = "CREATE TRIGGER IF NOT
        EXISTS changeIDTrigger" +
        " AFTER INSERT ON Person" +
        " BEGIN" +
        " UPDATE IDTable SET id = id + 1 WHERE 'tableName' = 'Person'; " +
        " UPDATE Person SET id = (SELECT deviceID + (SELECT id FROM IDTable
                WHERE tableName = 'Person')
                FROM DeviceID WHERE id = '1') WHERE id = new.id; " +
        " END;" +
        ");";
```

List 4.2: Trigger for assigning a correct id to new rows.

Since this amendment of ID should only occur when the inserted rows originates from the client and not when retrieving data from the server, a third table is used to keep track of the current state, so the trigger is only fired when synchronization is not in progress.

While this does restrict the use of the client during update, this is not believed to by of greater concern at the moment.

4.2 Implementing SAMD Synchronization

Step one and three of the SAMD algorithm are implemented as stored procedures on the server's MySQL database. Step two and four are implemented as an application on the client device.

4.2.1 Problems with MySQL

Because of the fact that MySQL in some ways deviates from the SQL standard, and that the stored procedures are database specific, we encountered a few problems implementing the algorithm.

One of the problems we encountered working with stored procedures in MySQL, was the inability to assign an SQL result set to a variable. This limited the way we could use stored procedures to divide a computation into logical cohesive parts.

An alternative way to send result sets between stored procedures, is to create a temporary table to store the different values in, as shown in listing 4.3.

```sql
DELIMITER //
DROP PROCEDURE IF EXISTS Callee;
CREATE PROCEDURE Callee ()
BEGIN
  INSERT INTO tmptable SELECT id, MD5(description) FROM stuff;
END //

DROP PROCEDURE IF EXISTS Caller;
CREATE PROCEDURE Caller ()
BEGIN
  CREATE TEMPORARY TABLE tmptable (id int, hash char(32));
  CALL Callee();
  SELECT * FROM tmptable;
END //
```

List 4.3: An example of returning dynamic results.

Another problem with the way MySQL implements stored procedures, is the inability to use dynamic SQL with stored procedures. This means that it is not possible to insert the value of a variable into an SQL statement.

We solved this problem by concatenating the SQL with the variables and wrapping them in a prepared statement, as shown in 4.4.

```sql
DELIMITER //
DROP PROCEDURE IF EXISTS runner;
CREATE PROCEDURE runner ()
BEGIN
  DECLARE tablename VARCHAR(20);
  DECLARE minID INT;
  SET tablename = 'mytable';
  SET minID = 4;

  CALL prepStmt(CONCAT('SELECT *
                                FROM ', tablename,
                               'WHERE id > ', myval));
END //

DROP PROCEDURE IF EXISTS prepStmt;
CREATE PROCEDURE prepStmt (IN query VARCHAR(1000))
BEGIN
  SET @query = query;
  PREPARE stmt FROM @query;
  EXECUTE stmt;
  DEALLOCATE PREPARE stmt;
  SET @query = null;
END //
```

List 4.4: An example of dynamic SQL.

In order to dynamically build a companion table, we needed a way to calculate the hash of all the rows in an arbitrary table. Instead of statically defining the column names of each table, we wrote a stored procedure that returns a comma-separated string with the column names, as shown in 4.5.

```
DROP PROCEDURE IF EXISTS ColumnNamesToString;
CREATE PROCEDURE ColumnNamesToString (IN dbname VARCHAR(20),
                                      IN tablename VARCHAR(20),
                                      OUT result VARCHAR(60))
BEGIN
  DECLARE done INT;
  DECLARE colname VARCHAR(20);
  DECLARE cols VARCHAR(60);
  DECLARE curCol CURSOR FOR (SELECT COLUMN_NAME
                            FROM INFORMATION_SCHEMA.COLUMNS
                            WHERE TABLE_SCHEMA=dbname
                            AND TABLE_NAME=tablename);
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done=1;
  OPEN curCol;
  SET done = 0;
  SET cols = '';
  REPEAT
    FETCH curCol INTO colname;
    #concat column names
    IF cols = '' AND done=0 THEN
      SET cols = colname;
    ELSEIF done=0 THEN
      SET cols = CONCAT(cols, ', ', colname);
    END IF;
  UNTIL done = 1
  END REPEAT;

  SET result = cols;
  CLOSE curCol;
END //
```

List 4.5: A stored procedure returning a CSV string containing column-names for a table.

4.2.2 Helper Functions

To simplify the synchronization procedures, we implemented the two procedures BuildCT and BuildCTWithName. The first procedure builds a companion table for a datatable and gives it a default name. The second procedure also generates a companion table for the datatable, but allow the caller to specify what the companion table should be named. If either one has the parameter *temporary* set to 1 the procedures only build a temporary table. These procedures are shown in listing 4.6.

```
DROP PROCEDURE IF EXISTS BuildCT;
CREATE PROCEDURE BuildCT (IN dbname VARCHAR(20),
                           IN tablename VARCHAR(20),
                           IN temporary TINYINT,
                           IN mid INT)
BEGIN
  DECLARE ctName VARCHAR(60);
  SET ctName = CONCAT(tablename, 'CTserver');
  CALL BuildCTWithName(dbname, tablename, temporary,mid,ctname);

END //

DROP PROCEDURE IF EXISTS BuildCTWithName;
CREATE PROCEDURE BuildCTWithName (IN dbname VARCHAR(20),
                                   IN tablename VARCHAR(20),
                                   IN temporary TINYINT,
                                   IN mid INT,
                                   IN CTName VARCHAR(20))
BEGIN
  DECLARE cols VARCHAR(60);
  DECLARE query VARCHAR(500);
  SET query = CONCAT('DROP TABLE IF EXISTS ', CTName);
  CALL prepStmt(query);
  CALL ColumnNamesToString(dbname, tablename, cols);

  SET query = CONCAT('SELECT id, MD5(CONCAT(',cols, '))
        AS hash FROM ', tablename);

  IF temporary=1 THEN
    SET query = CONCAT('CREATE TEMPORARY TABLE ', CTName, ' ', query);
  ELSE
    SET query = CONCAT('CREATE TABLE ', CTName, ' ', query);
  END IF;

  CALL prepStmt(query);
  SET query = CONCAT('ALTER TABLE ', CTName, ' ADD COLUMN (flag INT)');
  CALL prepStmt(query);
  SET query = CONCAT('ALTER TABLE ', CTName, ' ADD COLUMN (mid INT)');
  CALL prepStmt(query);

  SET query = CONCAT('UPDATE ', CTName, ' SET flag = 1, mid = ', mid);
  CALL prepStmt(query);
END //
```

List 4.6: The procedures for creating companion tables.

5 Summary

5.1 Conclusion

During this sprint, we have been working on implementing the two different synchronization approaches. However, we were unable to complete neither TSync nor SAMD during this sprint, which means that we will have to push some of our goals into the next sprint. The reason for our delay were, primarily, an increase in changes made on project-dependencies. These changes necessitated refactoring of the synchronization library. A detailed description of all external changes affecting our project will be given in section 1 of the *Perspective*-chapter.

What we did complete, was a detailed strategy, or policy, for handling various scenarios where a conflict or error might occur, such as merge conflicts and connection disruptions. We concluded which criteria was relevant for testing and comparing the two approaches. These criteria should have been used to collect data about TSync and SAMD, but as a result of them not being fully implemented, testing was postponed. Lastly, the application for running the tests, based on the previously mentioned criteria, was written and is ready to be utilized.

5.2 Reflection on Development Method

We tried to have a stand-up meeting on a daily basis, but due to poor discipline and scattered lectures, we failed to do so. It is our hope that once lectures subside, we will be able to improve on this practice. The choice made last sprint, for skipping estimation of workload while maintaining overview, seems to work well. The high dissimilarity of our tasks make it hard to carry over experience from previous sprints. Therefore, we don't feel like we would have a chance of realistically estimating the larger tasks, and estimating the smaller tasks would be a waste of time.

Part V

Fourth Sprint

1 Introduction

The purpose of this section is to give an overview of the chosen project for the current sprint. A description of the project's functionality and state will lead to a set of possible problems for the group to solve. These problems will be explained and lastly, a reasonable and feasible subset of problems will be chosen as the focus of the sprint.

1.1 State of Affairs

For the fourth sprint the group chose to continue working on the synchronization problem. The project at this time consists of; a working MySQL database on the server; a working SQLite database for each client; and, an android service allowing the clients to update themselves to fit changes in the server database. Implementation of the two synchronization approaches TSync and SAMD have begun, but are currently not in a complete or functional state. Criteria for testing and comparing the two approaches have been analyzed and explained in detail. Furthermore, policies for handling different problematic situations that might occur when synchronizing have been elaborated. An application for collecting test-data have been written and is ready for use once a synchronization library is available.

1.2 Current Problems

As in the third sprint, the major problems, or tasks, is to finalize TSync and SAMD so testing can commence. Once testing has been concluded, the collected data will be analyzed to decide on which synchronization approach to implement for the customers. We also haven't implemented our ID-handling completely yet.

1.3 Sprint Goal

To deliver a complete solution to the customers, it is important that we reach our final goals from the previous sprint, which are:

- Complete implementation of TSync
- Complete implementation of SAMD
- Run test and collect data
- Compare results and select final implementation

2 Implementation

2.1 Implementing SAMD Synchronization - Part 2:2

Now that we have defined all the helper procedures, we are finally able to construct the procedures that perform the synchronization steps of the algorithm.

2.1.1 Main Synchronization Steps

The first step in the algorithm is to perform a synchronization of the server datatable and its companion table. This is performed by the stored procedure SyncOne, as shown in listing 2.1.

```
DROP PROCEDURE IF EXISTS SyncOne;
CREATE PROCEDURE SyncOne (IN dbname VARCHAR(20),
                          IN tablename VARCHAR(20),
                          IN mid INT)
BEGIN
  DECLARE tableExist TINYINT;
  DECLARE ctName VARCHAR(60);
  DECLARE tmpTableName VARCHAR(60);
  DECLARE tmpTableCTName VARCHAR(60);
  DECLARE query VARCHAR(1000);

  SET ctName = CONCAT(tablename, 'CTserver');
  SET tmpTableName = 'tmptable';
  SET tmpTableCTName = CONCAT(tmpTableName, 'CTserver');
  SET query = "";

  CALL tableExists(tableExist, dbname, ctName);

  IF tableExist = 0 THEN
    # build new CT for current table.
    CALL BuildCT(dbname, tablename, 0, mid);
  END IF;

  # calculate new hash values for the datatable and insert them into
  # the corresponding ct + setting the flag positive.
  CALL BuildCTWithName(dbname, tablename, 1, mid, tmpTableCTName);

  SET query = CONCAT('UPDATE ', ctName, ' AS M, ', tmpTableCTName, ' AS T
                    SET M.hash = T.hash, M.flag = 1
                    WHERE M.id=T.id AND M.hash <> T.hash AND M.mid=',mid);
  CALL prepStmt(query);

  # build temporary table for the next query.
  CALL prepStmt(CONCAT('DROP TABLE IF EXISTS ', tmpTableName));

  SET query = CONCAT('CREATE TEMPORARY TABLE ', tmpTableName, ' (SELECT M.id
                    FROM ', ctName, ' AS M LEFT OUTER JOIN ', tablename, '
                      AS T
                    ON M.id = T.id
                    WHERE T.id IS null)');
  CALL prepStmt(query);

  # set hash values to null if the row is deleted in table.
  SET query = CONCAT('UPDATE ', ctName, '
                    SET hash = null, flag = 1
                    WHERE id IN  (SELECT * FROM ', tmpTableName, ')');
  CALL prepStmt(query);

  # create entries for newly created rows in the datatable.
  CALL prepStmt(CONCAT('DROP TABLE IF EXISTS ', tmpTableName));

  SET query = CONCAT('CREATE TABLE ', tmpTableName,
         ' (SELECT * FROM ', tablename, ' AS Z
          WHERE Z.id IN (SELECT T.id
                         FROM ', ctName, ' M RIGHT OUTER JOIN ',
                              tablename, ' T
                         ON M.id = T.id
                         WHERE M.id IS null))');
  CALL prepStmt(query);

  # build a temporary ct containing the values of the new rows.
  CALL BuildCTWithName(dbname, tmpTableName, 1, mid, tmpTableCTName);

  CALL prepStmt(CONCAT('DROP TABLE IF EXISTS ', tmpTableName));

  # insert newly hashed rows from tmpTableCT into the tablenameCT.
  SET query = CONCAT('INSERT INTO ', ctName, ' (SELECT *
                     FROM ', tmpTableCTName, ' )');
  CALL prepStmt(query);
END //
```

List 2.1: The implementation of the first step of the optimized SAMD algorithm.

The second step is performed on the client which then uploads a companion table to the server. At this point the server has two up-to-date companion tables for the current datatable and is now able to calculate which changes should be performed on the client- and server-tables. This is done by the procedure SyncThree which uses a full outer join to determine which action should be performed on a row by looking at their flag- and hash-values. The procedure, as shown in listing 2.2, stores the action for each row in a results table, so the client can use it to determine which action should be taken for the corresponding datatable row.

```
DROP PROCEDURE IF EXISTS SyncThree;
CREATE PROCEDURE SyncThree (IN dbname VARCHAR(20),
                            IN tablename VARCHAR(20),
                            IN mid INT)
BEGIN
  DECLARE serverCTName VARCHAR(20);
  DECLARE clientCTName VARCHAR(20);
  DECLARE resultsTableName VARCHAR(20);
  DECLARE tmpTableName VARCHAR(20);

  SET serverCTName = CONCAT(tablename, 'CTserver');
  SET clientCTName = CONCAT(tablename, 'CTclient');
  SET resultsTableName = CONCAT(tablename, 'results');
  SET tmpTableName = 'tmptable';

  # clientResults table schema (id, result - overwrite,fetch,conflict,delete)
  CALL prepStmt(CONCAT('CREATE TEMPORARY TABLE ', resultsTableName, '(id INT,
        action VARCHAR(20))'));

  # Full outer join serverCT and clientCT
        with columns id, S.flag, C.flag, S.hash, C.hash
  CALL prepStmt(CONCAT('CREATE TEMPORARY TABLE ', tmpTableName,
                      ' SELECT S.id AS id,
                               S.flag AS Sflag,
                               S.hash AS Shash,
                               C.flag AS Cflag,
                               C.hash AS Chash
                        FROM ', serverCTName, ' S LEFT OUTER JOIN ',
                                clientCTName, ' C ON S.id=C.id',
                                ' UNION ',
                       'SELECT S.id, S.flag, S.hash, C.flag, C.hash
                        FROM ', serverCTName, ' S RIGHT OUTER JOIN ',
                                clientCTName, ' C ON S.id=C.id'));

  CALL prepStmt(CONCAT('select * from ', tmpTableName));

                        ...    # code has been folded.

  # row is deleted on server
  # action: delete on local db.
  CALL prepStmt(CONCAT('INSERT INTO ', resultsTableName, '(id, action) '
                      'SELECT id, "LocalDelete"
                       FROM ', tmpTableName,
                     ' AS T WHERE Sflag=1
                              AND Shash IS null
                              AND Cflag=0'));

                        ...    # code has been folded.

END //
```

List 2.2: The implementation of the third step of the optimized SAMD algorithm.

The final part of the algorithm is run on the client, by executing SQL statements that perform the actions, described in the results table, on the remote and local databases. When a row has been synchronized its flag is set to 0 in all the companion tables unless it is a deletion in which case it is removed.

3 Summary

3.1 Status

The purpose of this section is to explain the state of the two synchronization approaches, i.e. how far from being finished they are.

3.1.1 Timestamp Synchronization

The partial solution using timestamps, as described in section 4.1 of the second sprint, have been integrated into the multi-project. This solution was chosen to allow the customers limited functionality, rather than no functionality, after the end of the semester.

3.1.2 Synchronization Algorithm based on Message Digest

All the server-side functionality has been implemented as SQL-scripts, as described in section 2.1. This leaves all the client-side functionality, and integration with the multi-project, to be done.

3.2 Conclusion

The external changes, which became influential in previous sprint, intensified due to higher productivity in the other groups. This required more collaboration with other groups on refactoring the existing partial synchronization implementation. Only minor advances were made as a result, leaving neither TSync or SAMD completed. For this reason, the testing-phase, for collecting and comparing results in the two approaches, were impossible and consequently omitted from the report.

A detailed description of all external changes affecting our project will be given in section 1 of the *Perspective*-chapter.

In conclusion, none of the sprint goals were reached. However, the required refactoring were completed and TSync in it's current state, with one-way synchronization, was integrated with the multi-project. That is, clients are able to request and receive data from the server, but any changes made on a client can not be sent to the server.

3.3 Reflection on development method

We decided to only have stand-up meetings whenever someone had something that needed the attention of everyone in the group, since the general practice lacked the support of the group. The effects of not being able to estimate our work assignments properly became a problem, as we had no clear estimation of our velocity, thereby lacking an idea of how far along we were with the implementation. This resulted in, us realizing too late in the sprint, that we couldn't complete our sprint goals.

Part VI

Discussion and Conclusion

1 External Requests and Firefighting

This section will contain a complete description of all the additional work, requested by external sources, delaying implementation of our project. Additionally, any other influential requests from other groups will be described.

1.1 Renaming Of Attributes

Early in the semester, one group found it necessary to have us rename specific attributes, as the attribute-names overlapped with keywords in Java. Likewise, during the final weeks of sprint four, the group, handling requests between applications and client database, renamed a large chunk of the client's database. As a result, we had to alter the server database to match these changes. We encountered a problem though, as the server contained about 42k entries of client data, which was not just dummy-data. The problem was that many primary keys, referenced to by other foreign keys, was renamed. So, in order to rename the primary keys, we first had to remove any foreign key constraints referencing the primary key, rename the primary key and finally re-implement the foreign key constraint. Likewise, in the synchronization library, we had to match all the changes made.

1.2 Structural Changes

Throughout the semester, we have received numerous requests to add or change the structure of the server database. One of the earliest requests was to change a foreign key, which pointed to a wrong table. Performing this change on both client and server luckily didn't have any effect on our implementation. Another group was working on building a system for handling sequences of *Pictograms*, and needed us to run the necessary changes to support storing these sequences. As these changes added additional content to the databases, we had to make sure our synchronization process included the new tables.

1.3 Duplicating Databases

One of the other groups was implementing a content management system and was in need of a completely identical and separate database to simulate and run tests on. Duplicating the database was easy enough, but during the semester we had to make sure any structural changes made in the primary database was reflected in this separate database.

1.4 Creating And Restoring Backups Of Client Data

Once the client's data was saved in the server database, we were tasked with creating a backup of the data and restore it if ever necessary. This did become necessary several times, as the primary database lost it's content for unknown reasons.

2 Perspective

2.1 Development Method And Practices

Our final thoughts on the practices used this semester are, that it is important to coordinate global-changes in the multi-project through the weekly SCRUM meetings, and to use continuous-integration throughout the development process. As many of the groups in the multi-project are dependent on certain other projects, using a shorter release cycle for these groups, will ease the integration and refactoring process needed once changes have been performed. Furthermore, having these dependency groups stop development at an earlier stage would allow the rest of the groups to incorporate these final changes.

For example, we experienced that one of our dependencies waited until the end of sprint 4, before integrating it's changes, causing our project, and many others to fail, leaving us little time to resolve the problems. To complement continuous-integration, test-driven development could be useful, for easier and faster refactoring.

Another problem was the lack of experience. When looking at Boehm and Turner's agility radar diagram [16] it is clear that the multi-project would be better served by using a more traditional plan-driven development method. In particular, too many developers where inexperienced in using and developing for the Android platform, and using agile development. Furthermore, the size of the development team being close to 60 developers is also suggest a more traditional method. As a consequence, we had the following problems:

- limited or poor information gathered from customers
- poor structure on meetings and information sharing
- poor execution of practices

These problems became less severe as the project progressed and people became more experienced. By the time we started improving, two sprints had already passed.

All in all, communication across the multi-project and between groups was the most important aspect and once we got past the initial barriers, communication was handled reasonably.

3 Further Development

3.1 Necessary Improvements

The necessary improvements are what is needed to complete the customer requirements.

3.1.1 Completion of TSync and SAMD

Obviously, the most pressing task is to finish a working solution of either TSync or SAMD. Finishing just one of the approaches in a completely functional state is highly desirable for the customers.

3.1.2 Commence Testing

Once both approaches have been completed, a testing-phase for collecting data based on the criteria already described, in section 2.4 of sprint 3, can begin. The purpose of the testing-phase is to compare the two approaches and choose which to integrate with the multi-project.

3.1.3 Integrate Solution

Once the best solution is chosen, it will need to be integrated, by collaborating with another group to call the initialization of the synchronization process. At the moment, the group responsible for this, is the "Launcher" group. The group is responsible for giving access to all other applications part of the multi-project. The synchronization process will be run before access to the other application are given, thereby making sure the content is synchronized and up-to-date before it is used.

3.2 Future Improvements

The future improvements are ideas to better the solution in various ways.

3.2.1 Improving Security

As the database will contain sensitive information, in form of contact information and images, it is important that the security of the synchronization process will be improved. The first part would be to encrypt all the data stored in the databases, and encrypt the connection between server and client, making the information harder to obtain for foreign parties.

3.2.2 Compressing Data

To improve the speed of the synchronization process, the data could be compressed before being sent. However, another testing-phase would be needed to conclude when it's reasonable to spent time compressing data rather than simply sending it. Smaller data-updates may be faster to simply send uncompressed, whereas large updates probably are wise to compress.

3.2.3 Changing DBMS

Developing synchronization on MySQL have proven difficult, but not impossible. If MySQL would be replaced by PostgreSQL, there might be minor improvements in speed as the complexity of implementing synchronization decreases. But this is just a guess, and would have to be analyzed further.

3.2.4 Limiting Data Transmissions

To minimize the transfer-size when synchronizing, it could be an idea to only download relevant data. For example, it might not make sense to download data pertaining to other departments or even other users. This would require multiple steps of synchronization as data for specific users are only available after logging in This would also limit the distribution of sensitive data.

4 Conclusion

Although we were unable to implement a complete solution, we improved the synchronization project significantly compared to its previous version. Going from a stand-alone solution working on an incorrect schema, to having a working one-way solution integrated as part of the multi-project.

Besides the solution mentioned, we have explored alternatives that, once implemented, are able to provide full two-way synchronization.

During the multi-project, we had a high degree of collaboration with the other groups, in particular with groups dependent on the database.

Part VII

Shared Activities

1 Database

1.1 The Structure of the Databases

The databases of the multi-project environment is structured as shown in figure 1.1. The main database is Remote DB, which each device synchronizes against. This enables users to log in at any device and have up-to-date information about itself. The local database on each device is LocalDB, which applications use to store and retrieve information from. It is the task of Sync-lib to facilitate synchronization between the Remote DB and the Local DB, which can be initiated from a specific application, illustrated as App$_{Sync}$. This construction has been created to ensure that the multi-project environment works even if the device is offline. The synchronization should be automatic.

To communicate with the local database, the applications in the multi-project environment use the Oasis-library, which is an API to the local database. This has been created to ease the programming for the applications.

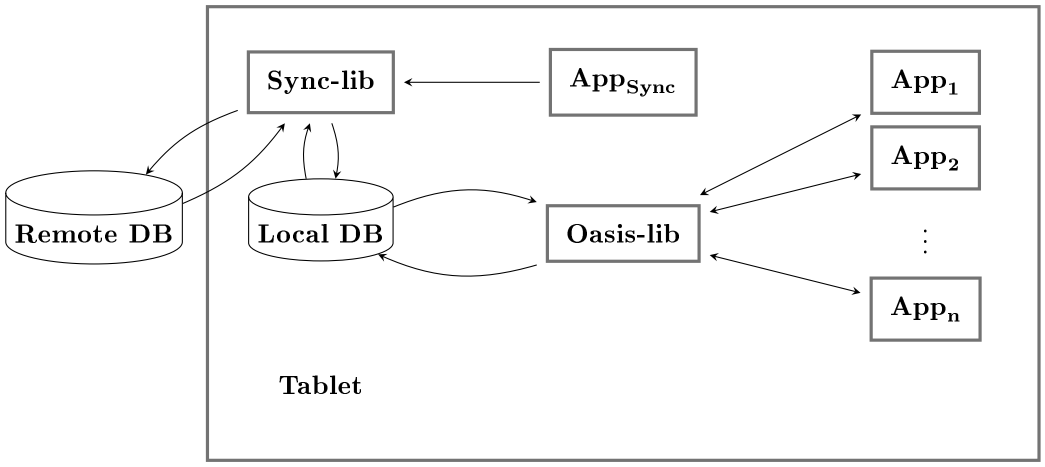


Figure 1.1: Overview of the Database Structure

1.2 Local database

As the applications run on the Android system, an obvious choice of database is SQLite, as it is fully supported on Android [17]. SQLite was also used by the previous database groups. An excellent tutorial for using SQlite is written by Vogel [18], describing the use of both SQLite and `ContentProviders`.

SQLite

SQLite is a public domain, light-weight, embedded SQL database engine [19]. It implements most of SQL-92 [20], but omits a few advanced features [21].

Implementation

The implementation of LocalDB follows the recommended approach for creating an SQLite database [17]. For other applications, to use the database, a class extending `DbProvider` [22] was implemented, giving access to the standard CRUD operations.

1.3 Remote database

The purpose of the remote database is to act as a data-central from where clients can synchronize their local database by sending and receiving updated content.

Database Management Systems

The system is currently running on MySQL, which is adequate for our purposes at the moment. However, if it is found that MySQL does not support needed features, another DBMS, such as PostgreSQL or Oracle, could be considered. It should be noted that Oracle is not free to use.

The Schema

The existing schema was created by students from previous semesters at Aalborg University [1]. The schema is in Boyce-Codd normal form.

Pictogram

Pictogram stores image data with associated sounds and descriptions.

- Every *Pictogram* can have associated with it an author, which is the administrative *User* who created it.
- A many-to-many relation to *Tag* exists through the relation-table **pictogram_tag**.

Category

Category is used to group pictograms.

- A *Category* can be a child of another *Category*, specified by the foreign key **super_category_id**.
- A many-to-many relation to *Pictogram* exists through the relation-table **pictogram_category**.

Department

Department is used to group profiles by affiliation.

- A *Department* can be a child of another *Department*, specified by the foreign key **super_department_id**.
- Every *Department* can have associated with it an author, which is the administrative *User* who created it.
- A many-to-many relation to *Pictogram* exists through the relation-table **department_pictogram**.
- A many-to-many relation to *Application* exists through the relation-table **department_application**.

Profile

Profile stores relevant information about persons, such as contact information and roles.

- A *Profile* is connected to a specific *Department* through the foreign key **department_id**.
- A *Profile* can be associated with a *User* through the foreign key **user_id**.
- Every *Profile* can have associated with it an author, which is the administrative *User* who created it.
- A many-to-many relation to *Pictogram* exists through the relation-table **profile_pictogram**.
- A many-to-many relation to *Category* exists through the relation-table **profile_category**.
- A many-to-many relation to itself exists through the relation-table **guardian_of**.

User

User stores login information and is used to gain access to the system.

- A many-to-many relation to *Department* exists through the relation-table **admin_of**.

Application

Application stores meta information about applications and is used to control access to applications.

- Every *Application* can have associated with it an author, which is the administrative *User* who created it.

Tag

Tag is used to ease searching for pictograms.

1.4 Structure of Oasis Lib

The structure of Oasis consists of models, controllers and metadata. The responsibility of the models and controllers is determined by following the Model-View-Controller pattern.

Model

The model describes the structure and handles the logic of the system. Change of data is handled in the model, and in case of major changes of data, the view is updated to display the current data.

View

The view displays the data, and it is used to interact with the system. The view interacts with the controller to request existing data and to modify or create new data.

Controller

The controller is the communication medium between the model and the view. It is only possible to access and modify data through the controller.

Implementation

In Oasis there are only models and controllers. Oasis is used to communicate with the database, and therefore it has no graphical interface and hence no view. The views are represented by each application in the multi-project environment. Metadata in Oasis is used to specify the information about the database, and it is not a part of the Model-View-Controller pattern.

Figure 1.2 shows the design of our implementation.

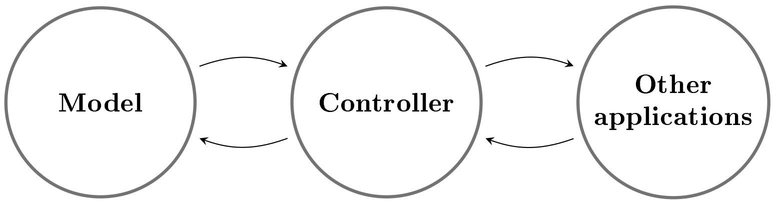


Figure 1.2: The structure of Oasis lib

2 Server

As part of the multi-project one of our group members had the responsibility for maintaining the multi-project's server. Over the course of the semester the server has been assigned a variety of responsibilities e.g. build-server, web-server, git repository controller, project management, database provider, etc. In this chapter we will describe how these features has been implemented. The server has a 3Ghz processor, 1GB ram and 65 GB disc space. It runs *Ubuntu Linux* v. 13.10.

2.1 Database

The server runs MySQL which contains the databases responsible for serving data to *Redmine*, a project management system, and the multi-project applications. The database for *Redmine* is simply called *Redmine* and is maintained by the *Redmine* web-application and it's associated bot user. The multi-project applications use a common framework that access the server database through a user called **sqluser**. During the semester, a number of other databases has been added which contain test data. The purpose of having these databases are to prevent different groups from affecting each others tests during development.

2.2 Continuous Integration Using Jenkins

One of the primary tasks of the server is to support continuous integration. To perform this function the server runs a framework called *Jenkins* which provides a way to easily manage build-scripts using a web front-end. The framework also provides a number of plugins, that makes it possible to support other technologies like *Git*, e-mail and *Gradle*, as well as different build-strategies and build-script managements tools.

2.2.1 Migration To Gradle

The server was initially configured to build using a tool called *Ant*, which is well integrated into the *Eclipse IDE*. However, during the first few weeks of the multi-project the groups decided to switch development to *Android Studio* which instead uses *Gradle* build-scripts.

Problems

As *Jenkins* executes all build-scripts from command-line, it is important that all project build-scripts are able to do this in a non-interactive way. This means that the *Gradle* scripts cannot require user input. Most groups use *Android Studio* to compile their projects, which ignores one specific type of errors, known as *lint*-errors by default. This became a problem for the server as the *Gradle* build-tool fails on *lint*-errors by default. The solution was to add a directive to all the projects *build.gradle* files that make *Gradle* ignore *lint*-errors.

Although the groups had decided to use *Gradle* version 1.9 to compile all the projects, a lot of them ended up using version 1.10 as a side-effect of an update to *Android Studio*. This meant that their build-scripts became incompatible with the version of *Gradle* that was running on the server. To help the groups use the right version, a guide was written on how to install *Gradle* 1.9 and correct the build-scripts to accept the downgrade.

2.2.2 Signing

Besides building the projects, it also became the responsibility of the server to sign the *Android Application Package Files* (APK) generated by *Gradle* so that they could be distributed using the *Google Play* platform. This was accomplished by generating a keystore using the *GNU keytool* and using the tool *Jarsigner* with it to sign the APKs. As a final step the files were renamed and moved to a directory where they can be accessed from the web:

cs-cust-06-int.cs.aau.dk/files/signed

To automate the signing process, a *Jenkins* project was created that first builds all the other projects and then runs a shell script that performs the actions described above.

2.3 Git and Gitolite

The server also functions as a centralized hub for all the *Git* repositories belonging to the different projects. To do this, the server runs the daemon-program *Gitolite* which control read/write access to the repositories.

Because *Git* itself is decentralized, deleting a repository needs to be carried out first using *Git* and then the repository files must be manually deleted on the server. If either step is omitted the repository will reappear.

The *Git* repositories are presented to the groups using the *Apache2* web-server. The URLs for the repositories are:

cs-cust06-int.cs.aau.dk/git/{projectname}

for read/write access, and

cs-cust06-int.cs.aau.dk/git-ro/{projectname}

for read-only access. Access to the repositories are authenticated using the *Apache2* web-server.

2.4 Web Server

Two of the groups participating in the multi-project, *Ugeskema* and *Webadmin*, implements a web-page as part of their project. Therefore, the server must also function as a web-server for these projects. The server also has to support the *Redmine* project management system and the web front-end to *Jenkins*.

To provide web access, the server runs the *Apache2* web-server, which is configured to authenticate using *LDAP* from an existing university server. Authentication is required for the *Redmine* site as well as the *Git* and *Jenkins* web-pages.

2.5 General Problems/Issues

Initially, *Jenkins* was configured to build a project every time something new was committed to the master-branch of its repository. This caused the server to be overloaded at times with high development activity. Therefore *Jenkins* was reconfigured so that a project only is being built if it is manually started by a user or as part of a daily global build for all project.

2.5.1 Server Resources And Logical Volume Manager

During the first sprint, it became apparent that the server would be unable to support the development and test data with the allocated disc-space. Therefore we petitioned AAU IT-Support to have the disc-space augmented with an additional 25 GB.

The server allocates its disc-space using *Logical Volume Manager* (LVM) which is a logical layer on top of normal disc partitions. LVM makes it easier to manage changing requirements for disc space. The server has a number of physical volumes which are assigned to one of two volume groups, *data* or *system*. The space from the two volume groups are assigned to four different parts of the system. The data-volume group is only assigned to *srv* which is mounted on /srv but the system-volume group is assigned to *log* mounted on /log, *root* mounted on / and *swap* which is assigned as swap space.

The additional 25 GB was assigned to the data-volume group.

Most of the data requirements for the server has been moved to a directory below /srv this includes the data-directory for the *MySQL* server and the home directories for all users. To prevent the system from becoming too difficult to administer, the movement has been implemented using symbolic links from the original location to the new. Therefore all default paths are preserved.

2.5.2 LDAP Server Overload

During the last sprint, the university *LDAP* server that *Apache* uses to authenticate, became overloaded causing access to be denied to *Jenkins* and *Git*. This was solved by reconfiguring *Apache* to use a different *LDAP* server.

2.6 Further Development

Since the server came online, a new long-term support version of *Ubuntu* has become available. To ensure that the server has the latest updates, it would make sense to upgrade to this version.

To prevent future confusion among developers, it would be a good idea to upgrade *Gradle* to the latest version used by *Android Studio*.

There exists several plugins for *Redmine*, such as e-mail notification, embedded video, javadoc, etc. which could improve its usefulness.

Bibliography

[1] B. Flindt, H.L. Magnussén, J.B. Tarp, and Simon Jensen. Wasteland. http://projekter.aau.dk/projekter/files/77170896/master.pdf.

[2] W3. Soap version 1.2 part 1: Messaging framework (second edition). http://www.w3.org/TR/soap12-part1/.

[3] MySQL.com. Timestamps in mysql, . http://dev.mysql.com/doc/refman/5.5/en/timestamp-initialization.html.

[4] sqlite.org. Timestamps in sqlite. http://sqlite.org/lang_datefunc.html.

[5] SQLite.org. Using triggers in sqlite, . https://www.sqlite.org/lang_createtrigger.html.

[6] Symmetricds.org. Symmetricds. http://www.symmetricds.org/doc/3.5/html/introduction.html#definition.

[7] MySQL.com. Using triggers in mysql, . http://dev.mysql.com/doc/refman/5.0/en/triggers.html.

[8] P. Leach, M. Mealling, and R. Salz. A universally unique identifier (uuid) urn namespace. http://tools.ietf.org/pdf/rfc4122.pdf.

[9] Mi-Young Choi, Eun-Ae Cho, Dae-Ha Park, Chang-Joo Moon, and Doo-Kwon Baik. A database synchronization algorithm for mobile devices. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5505945.

[10] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. Handbook of Applied Cryptography. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996. ISBN 0849385237.

[11] developer.android.com. Asynctask, . http://developer.android.com/reference/android/os/AsyncTask.html.

[12] MySQL.com. Data type storage requirements, . http://dev.mysql.com/doc/refman/5.0/en/storage-requirements.html.

[13] SQLite.org. Datatypes in sqlite version 3, . http://www.sqlite.org/datatype3.html.

[14] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. Database System Concepts (6th edition). McGraw-Hill. ISBN 0073523321.

[15] developer.android.com. Managing your app's memory, . http://developer.android.com/training/articles/memory.html.

[16] Boehm and Richard Turner. Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321186125.

[17] Android Developers. Storage options, . http://developer.android.com/guide/topics/data/data-storage.html#db.

[18] Lars Vogel. Android sqlite database and content provider - tutorial. http://www.vogella.com/tutorials/AndroidSQLite/article.html.

[19] SQLite. About sqlite, . http://www.sqlite.org/about.html.

[20] Sql-92. http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt.

[21] SQLite. Sql features that sqlite does not implement, . https://sqlite.org/omitted.html.

[22] Android Developers. Content providers, . http://developer.android.com/guide/topics/providers/content-providers.html.

A Appendix

A.1 Sprint 1 backlog

- On some update calls the server returns an OK, but no SQL is executed.
- If a user can access several profiles there is no way to see which settings belong to which profiles for any given application.
- SQL errors can crash the server because null-errors in some query results go unhandled.
- There is no way to make a user department administrator without directly accessing the database.
- There are no SQL transactions in the API which means race conditions are a risk.
- Encryption and hashing should be added to the database and the connections.
- It should be possible to create a copy of pictogram if an update is made to one that has multiple links.
- A search function should be implemented to allow the public setting on pictograms to let the pictogram show up in searches.
- The synchronization should be updated to have support for selective download from the database.
- A library for Android applications should be made to allow them to access the local database.
- The local database should not be allowed to exceed a certain size.
- Synchronization should be automatic instead of manual.
- A thread pool should be implemented to prevent thread over ow on the central server.
- Database information should not be hardcoded.
- Sessions should be implemented. Currently the sessions are not used for authentication, which was part of their intended purpose.
- Currently the views only accommodate selecting by user id, not by profile id. This means that when a guardian is logged in, even if the tablet is switched to the child's profile (which is a projected functionality of the GIRAF system), all pictograms, applications etc. that the guardian can access will also be accessible to the child. This should be fixed by adding a profile id column to the different views where users need it.

A.2 Scripts for implementing timestamp proof of concept

```
    @Override
    protected Object doInBackground(Object[] objects) {

        db = MainActivity.getDb();

        try{
            Class.forName("com.mysql.jdbc.Driver").newInstance();
            Connection conn = (Connection) DriverManager.getConnection(URL,
                USER, PASS);

            String result = "Database connection successful.\n";
            Statement st = (Statement)conn.createStatement();

            db.open();
            String lastSync = db.getLastSync();

            String sql = "SELECT * FROM Person " +
                    "WHERE timestamp > '" + lastSync +"'";

            ResultSet rs = st.executeQuery(sql);
            ResultSetMetaData rsmd = (ResultSetMetaData)rs.getMetaData();

            while (rs.next()){
                db.createPerson(rs.getInt(1), // ID
                                rs.getString(2), // name
                                rs.getString(3), // address
                                rs.getString(4)); //timestamp

                result += rsmd.getColumnName(1) + ":" + rs.getInt(1);
                result += rsmd.getColumnName(2) + ":" + rs.getString(2) +
                    "\n";
            }

            db.updatedSync();
            db.close();
        }
        catch (Exception e) {
            Log.d("Database", "Connection failed.");
            e.printStackTrace();
        }

        return null;
    }
```

List A.1: The method for downloading new entries in the database

```
    public void createPerson(int id,  String name, String address,String
        timestamp){
        String sql = "INSERT OR REPLACE INTO Person (id, name, Address,
            timestamp)" +
                "VALUES (" + "'" + id + "', '" + name + "', '" + address +
                    "', '" + timestamp + "')";

        sqLiteDatabase.execSQL(sql);
    }
```

List A.2: Inserting the new data into the database

```
    public String getLastSync(){
                /*Default time stamp in case the tablet never before have
                    tried to synchronize. */
        String lastSync = "1970-01-01 00:00:00";

        try{
            String sql = "SELECT * FROM LastUpdate;";

            Cursor curLastSync = sqLiteDatabase.rawQuery(sql,null);

            if(curLastSync != null){
                curLastSync.moveToFirst();
                lastSync = curLastSync.getString(1);
            }
        }
        catch (Exception e){
            e.printStackTrace();
        }
        return lastSync;
    }
```

List A.3: Retriving the last time the client was synced

```
    public void updatedSync(){
        String sql = "INSERT OR REPLACE INTO LastUpdate (id, timestamp)" +
                " SELECT '1', datetime('now', 'localtime');";
        sqLiteDatabase.execSQL(sql);
    }
```

List A.4: Inserting the updated synchronization time in the first row

A.3 Testing Criteria: Time

Data	Relations	Total attempts	Worst time	Best time	Avg. time
No changes	None				
Small	Few				
	Many				
Medium	Few				
	Many				
Large	Few				
	Many				

Table A.1: Test results for **Time** when using approach X

A.4 Sequence Diagram

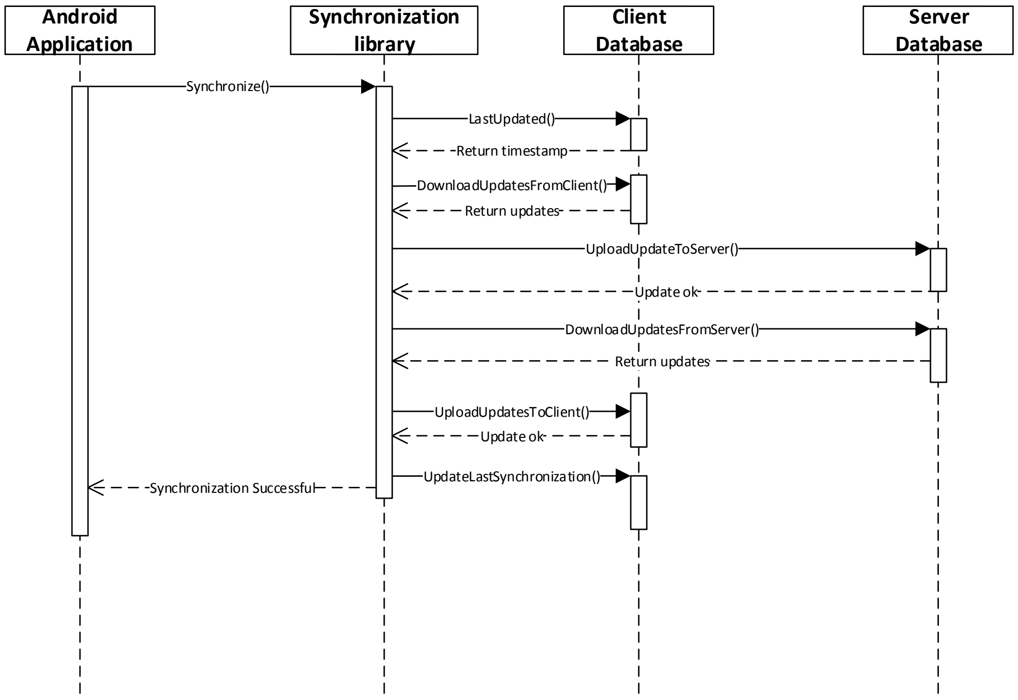


Figure A.1: The new sequence diagram for TSync