Hochschule RheinMain
Fachbereich Design Informatik Medien
Studiengang Angewandte Informatik

Bachelor-Thesis
zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

BIBLIOTHEK ZUR DATENBANK SYNCHRONISIERUNG FÜR MOBILE ANWENDUNGEN

---

vorgelegt von Tobias Reichert <mail@teamtobias.de>

am 19. Oktober 2017

Referent: Prof. Dr. Panitz
Korreferent: Prof. Dr. Martin

Erklärung

Ich versichere, dass ich die Bachelor-Thesis selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Ort, Datum | Tobias Reichert

Hiermit erkläre ich mein Einverständnis mit den im Folgenden aufgeführten Verbreitungsformen dieser Bachelor-Thesis:

Verbreitungsform	**ja**	**nein**
Einstellung der Arbeit in die Hochschulbibliothek mit Datenträger		√
Einstellung der Arbeit in die Hochschulbibliothek ohne Datenträger	√	
Veröffentlichung des Titels der Arbeit im Internet	√	
Veröffentlichung der Arbeit im Internet	√	

Ort, Datum | Tobias Reichert

Inhaltsverzeichnis

Zusammenfassung

Diese Arbeit behandelt die Konzeption und Entwicklung eines Prototypen zur Kopplung von Datenbanken im mobilen Kontext. Dabei wird ein besonderes Augenmerk auf das Benutzen von schon etablierten Datenbanksystemen gelegt. Das fertige Projekt enthält eine Bibliothek, die von einer Anwendung genutzt werden kann, um ihre Daten auf einen Server zu replizieren. Auch der Server, der das Gegenstück zur Bibliothek bildet, ist Teil des Projekts. Diese beide Komponenten beinhalten keine eigene Datenbank, sondern können über eine Schnittstelle an unterschiedlichste Datenhaltungssysteme angeschlossen werden. Damit kann der Nutzer immer von mehreren Geräten auf seine Daten zugreifen, auch wenn keine Verbindung zum Internet besteht. Eine Fallstudie begleitet, in Form einer Notiz-App, diese Bachelor-Thesis mit dem Titel: *Bibliothek zur Datenbank Synchronisierung für Mobile Anwendungen*.

Abstract

This work deals with the conception and development of a prototype for linking databases in the mobile context. Particular attention is paid to the use of already established database systems. The finished project contains a library that can be used by an application whose data is to be replicated on a server. The server, which is the counterpart to the library, is part of the project. These two components do not have their own database, but can be connected via an interface and different data storage systems. This allows the user to have access to his data from several devices, even if there is no connection to the Internet. A case study accompanies this Bachelor thesis with the title: *Library for database synchronization for mobile applications* in the form of a note app.

Kapitel 1

Einleitung

1.1 Idee

Fast jeder Programmierer, der schon mal eine App geschrieben hat, stand vor der Entscheidung: Die Daten direkt auf dem Gerät speichern oder besser im Internet ablegen? Beide Varianten haben Vor-, aber auch Nachteile.

Eine große Zahl von Nutzern etwa speichern ihre Einfälle gerne in Notiz-Apps ab. Leider stoßen sie dabei auf ein Problem. Denn liegen die Daten - in diesem Fall die Notizen - direkt auf dem Gerät, kann der Nutzer sie ohne Internetverbindung abrufen. Mit einer solchen Datenbank funktioniert eine Anwendung auch in Gegenden, in denen das Internet schlecht oder sogar gar nicht vorhanden ist. Sollte der Nutzer das Gerät allerdings verlieren, gehen auch alle Daten verloren. Und da die Daten nur an einem Ort gespeichert sind, kann der Nutzer auch nicht über ein anderes Gerät auf sie zugreifen. Denn das ist nur möglich, wenn die Daten auf einer externen Datenbank gehalten werden. Dann kann der Nutzer von jedem internetfähigen Gerät auf seine Daten zugreifen und sie können auch nicht verloren gehen. Der Haken: Ohne Internetverbindung kein Zugriff auf die Daten.

Beiden Lösungen fehlt also genau das, was die jeweils andere Idee auszeichnet. Das Ziel dieser Bachelorarbeit ist, dem Nutzer ein System zu bieten, das beide Vorteile vereint. Denn Notiz-Apps sind nur eine von sehr vielen Anwendungen - sowohl auf Computern, als auch auf mobilen Endgeräten - die von dieser Problematik betroffen sind.

Wer zu diesem Thema im Internet nach Lösungen sucht, stößt auf wenig Antworten. Zu finden sind nur Selbstbau-Methoden oder Systeme, die eine selbst gebaute Datenbank beinhalten. Doch die etablierten Datenbank-Systeme sind meist schon sehr gut optimiert.

Außerdem sind die Systeme so verbreitet, dass es bei Problemen nicht schwer ist, Hilfe zu finden. Eine völlig neue Datenbank wäre daher wenig sinnvoll.

Deshalb löst diese Arbeit das Problem mit nur einer Synchronisationsschicht zwischen der Datenbank auf dem Gerät und einer großen Datenbank im Internet, die die Daten synchronisiert. Somit können altbewährte und verschiedene Datenbanken miteinander gekoppelt werden. Diese Lösung möchte ich unter dem Namen "Bibliothek zur Datenbank Synchronisierung für Mobile Anwendungenïm Rahmen dieser Arbeit entwickeln.

1.2 Aufgabenstellung

In dieser Bachelor-Thesis soll ein Konzept entwickelt werden, dass zwei verschiedene Datenbanksysteme über ein nicht immer verfügbares Netzwerk synchronisiert. Zusätzlich soll ein Prototyp gebaut werden, der am Beispiel einer einfachen Notiz-App getestet wird.

1.3 Grundlagen

Bevor sich diese Arbeit der Frage widmet, wie zwei Datenbanken verbunden werden können, sollte zunächst auf die Theorie einer Datenbank eingegangen werden.

1.3.1 ACID

ACID, zu Deutsch AKID, ist ein Akronym, das in der Informatik mit Datenbanken zusammenhängt. Die Abkürzung steht für atomicity (Atomarität), consistency (Konsistenz), isolation (Isolation) und durability (Dauerhaftigkeit). Sie sind häufig erwünschte Eigenschaften von Verarbeitungsschritten in einer Datenbank. Die Prinzipien wurden von Theo Härder und Andreas Reuter im Jahr 1983 aufgestellt. Die Erkenntnisse dieser Arbeit sind heute in fast jedem modernen Datenbanksystem umgesetzt. Die einzelnen Eigenschaften besagen folgendes:

- **Atomarität** Eine Folge von Operationen kann zu einer atomaren Operation - meist Transaktion genannt - zusammengeführt werden. Diese Transaktion wird von dem Datenbankmanagementsystem (DBMS) wie eine einzige Operation behandelt, das bedeutet die Folge wird entweder vollständig oder gar nicht ausgeführt. Zwar werden die Teile der Operation immer noch nacheinander ausgeführt, doch werden die

Veränderungen erst ganz am Ende, wenn alles korrekt verlaufen ist, persistent geschrieben. Sollte im Laufe der Transaktion ein Fehler auftreten oder der Benutzer abbrechen wollen, kann der Ursprung vor der Transaktion mit einem Rollback wiederhergestellt werden.

- **Konsistenz** Die Konsistenzbedingung fordert die Integrität, also die Richtigkeit der Daten. Dabei gibt es zwei verschiedene Konsistenten: einmal die Konsistenz einer klassischen relationalen Datenbank und einmal die Konsistenz über ein gesamtes verteiltes System. Bei einer klassischen relationalen Datenbank geht es um die innere Konsistenz, die durch sogenannte Integritätsbedingungen festgelegt wird. Diese können zum Beispiel sein: Jedes Attribut muss im definierten Wertebereich liegen, der Primärschlüssel muss eindeutig und nicht null sein und der Fremdschlüssel muss auf ein existierendes Objekt zeigen oder null sein. Dabei können noch weitere Integritätsbedingungen von der Datenbank oder dem Benutzer festgelegt werden [Wik17c]. Für Konsistenz über ein verteiltes System siehe das CAP-Theorem 1.3.2.

- **Isolation** Operationen die gleichzeitig ausgeführt werden, müssen so gut es geht voneinander isoliert werden, um die Gefahr einer gegenseitigen Beeinflussung zu minimieren.

- **Dauerhaftigkeit** Das Prinzip der Dauerhaftigkeit fordert von einem System, dass es die Änderungen einer Transaktion nach ihrem erfolgreichen Ende dauerhaft speichert. Auch bei Systemfehlern, sowohl hard- als auch softwareseitig, dürfen die Daten nicht verloren gehen.

[Vos] [Wik17a]

1.3.2 CAP-Theorem

Im Jahr 2000 stellte der Informatiker Eric Brewer eine Vermutung über verteile Systeme auf. Dabei behauptete er, ein solches System könne niemals alle drei Eigenschaften - Consistency (Konsistenz), Availability (Verfügbarkeit) und Partition Tolerance (Ausfalltoleranz) - gleichzeitig garantieren. Lediglich zwei seien möglich:

- **(C consistency) Konsistenz** Die Konsistenz über ein ganzes verteiltes System fordert, dass alle Replikationen der Daten immer gleich sind.

- **(A availability) Verfügbarkeit** Alle Anfragen an das System werden immer in einer akzeptablen Antwortzeit beantwortet.

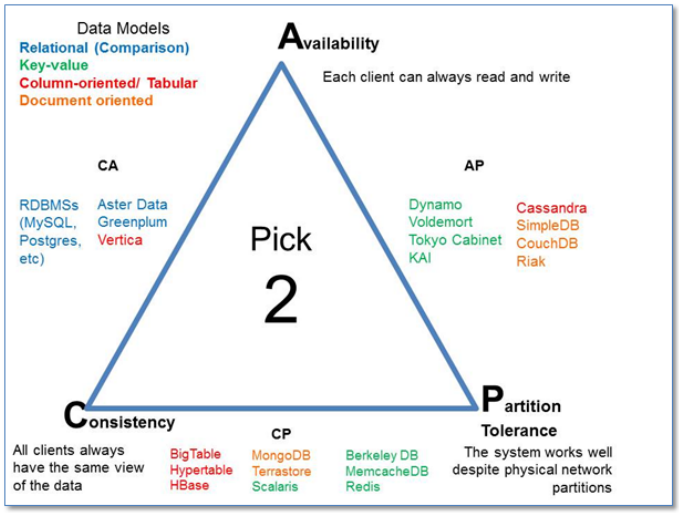

Abbildung 1.1: CAP-Theorem als Dreieck [Lab14]

- **(P partition tolerance) Ausfalltoleranz** Sollte das System sich in mehrere Partitionen teilen, kann es trotzdem weiter seine Anfragen bedienen.

Dabei sollten die Eigenschaften als Größen betrachtet werden. Antwortet das System schnell und immer, ist die Verfügbarkeit hoch - ist es langsam oder antwortet gar nicht, ist die Verfügbarkeit niedrig.

Zwei Jahre später konnten Seth Gilbert und Nancy Lynch Brewers Vermutung beweisen und sie als CAP-Theorem etablieren.

Wie in der Abbildung 1.1 zu sehen, wird das CAP-Theorem oftmals als Dreieck visualisiert, an dem sich der Programmierer eine Kante aussuchen kann. Somit ergeben sich drei Möglichkeiten: AP, CA und CP. Im Folgenden werden wir uns mit der Kombination AP auseinandersetzen, die oftmals im Zusammenhang mit eventually consisty zu hören ist. [Wik16] [Was16]

1.3.3 AP und Eventual consistency

Immer häufiger werden Anwendungen so groß, dass sie über mehrere Server verteilt werden müssen. Dabei werden die gleichen Datensätze auf mehreren Datenbanken gehalten, sogenannte Replikationen. Da bei solchen Systemen die Ausfalltoleranz und Verfügbarkeit sehr wichtig ist, sind die Replikationen untereinander nicht immer ganz konsistent. Das heißt, eine Änderung, die auf einem Server vollzogen wird, benötigt eine gewisse Zeit, bis sie auch auf alle anderen Server übernommen ist. In dieser Zeit sind die Kopien nicht identisch.

Oftmals ist diese Inkonsistenz nicht von großer Bedeutung. Bei einer Änderung eines DNS-Eintrags kann es etwa Tage dauern bis alle DNS-Server diese Änderung mitbekommen. Da die Verfügbarkeit und Ausfalltoleranz aber viel wichtiger ist (siehe CAP-Theorem 1.3.2), vernachlässigt man die Konsistenz. Zudem kommen diese Änderungen nur selten vor. [Wik16] [Wik17c] [Was16]

1.4 Genutzte Software

Um das Projekt umzusetzen, ist es nötig, sich auf einige Software-Pakete und Bibliotheken zu verlassen.

1.4.1 Java

Als Programmiersprache und Umgebung wird Java, beziehungsweise die Java Virtual Machine, benutzt. Sie bietet die Möglichkeit der Plattformunabhängigkeit, was gerade bei einer Bibliothek sehr wichtig ist. Zudem liefert Java selbst viele Bibliotheken und Erweiterungen mit. Das bedeutet insgesamt weniger Programmieraufwand als beispielsweise C gefordert hätte. Ein weiterer wichtiger Punkt ist, dass das mobile Betriebssystem Android eine Java-Laufzeitumgebung bietet [Wik17b]. Als Bibliotheken werden der MySQL- und SQLite-Connector genutzt, um sich mit den Datenbanken zu verbinden. Damit die Software auch auf Android laufen kann, wird auf Java 7 entwickelt.

1.4.2 Android

Mit 86,1 % Marktanteil im ersten Quartal 2017 ist Android das am meisten gekaufte Smartphone-Betriebssystem [VB17]. Um das Projekt auch auf dem mobilen Markt anzubieten, soll bei der Entwicklung darauf geachtet werden, dass die Client-Bibliothek mit Android kompatibel ist. Da Android-Applikationen auch mit Java geschrieben werden, ist der Aufwand dafür recht gering.

1.4.3 SQLite

Um die Daten auf den Geräten der Nutzer bereitzustellen wird SQLite benutzt. Wie der Name schon vermuten lässt, ist es eine SQL konforme Datenbank. Die Besonderheit dabei ist, dass es nur eine Bibliothek ist, die direkt in die eigentliche Anwendung intrigiert

wird. Dabei liegen die gesamten Daten in einer Datei. Somit gibt es auch keine Client-Server-Architektur und Rechte. Da diese Software aber sowieso nur auf einem Endgerät benutzt werden soll, kann auf diese Dinge ohne Einschränkung verzichtet werden. Zudem vereinfacht SQLite das Typensystem, indem es alle Daten in nur fünf Typen verpackt: NULL, INTEGER, REAL, TEXT, BLOB. [sql17]

Diese Programmbibliothek wird seit 2000 von Richard Hipp in C geschrieben und gepflegt. [Wik17e]

1.4.4 MariaDB

Um die Daten mehrerer Nutzer, Geräte und Anwendungen zu speichern, wird die Datenbank MariaDB genutzt. MariaDB ist ein freier, 1 : 1 kompatibler fork von MySQL [Mar17] und damit die am zweithäufigsten genutzte Datenbank [eng17]. Sie bietet eine Client-Server-Architektur, Rechteverwaltung [Wik17d] und ist sehr performant auch unter hoher Last [dig14].

Kapitel 2

Analyse

Es soll eine Bibliothek entwickelt werden, die eine Datenbank auf einem Anwendergerät mit einer Datenbank auf einem Server synchronisiert. Dabei soll der Nutzer der Bibliothek so wenig wie möglich von der Synchronisation mitbekommen. Nachfolgend werden die Anforderungen an die Bibliothek aufgelistet:

2.1 Vorbedingung

Das Projekt soll keine generelle Lösung für Datenbanksychronisierung sein, sondern es soll den Spezialfall von personenbezogenen Daten, die komplett auf dem Gerät gespeichert werden, abdecken. Dabei gibt es folgenden Bedingungen:

- Die komplette Tabelle wird Synchronisiert.
- Die kleinste Synchronisierungseinheit ist eine Tabellenzeile.
- Es gibt mehrere Benutzer pro Server.
- Es gibt mehrere Geräte pro Benutzer.
- Es gibt mehrere Tabellen pro Gerät.

2.2 Funktionale Anforderungen

Das Verb „muss“ beschreibt eine Vorgabe, die exakt umgesetzt werden muss.

Das Verb „soll“ beschreibt eine Empfehlung, meist gibt es alternative Lösungen, die zum selben Ergebnis führen.

Das Verb „kann“ beschreibt eine Möglichkeit, deren Erfüllung optional ist.

A.F.1 **Hauptaufgabe**
Das System muss zwei verschiedene Datenbanksysteme über ein nicht immer verfügbares Netzwerk synchronisieren.

A.F.2 **Automatische Synchronisation**
Der Programmierer muss sich um die eigentliche Synchronisation nicht kümmern.

A.F.3 **Server-Client-Architektur**
Das Projekt soll eine Server-Client-Architektur haben, die im Verhältnis 1:n steht.

A.F.4 **Lesen und schreiben**
Alle Geräte dürfen lesen und auch schreiben.

A.F.5 **Immer verfügbar**
Die Daten müssen auf allen Geräten immer verfügbar sein, auch wenn keine Verbindung zum Server besteht.

A.F.6 **Tabellenauswahl**
Das System soll eine Möglichkeit besitzen, in der die Anwendung, die das System benutzt, Tabellen zur Synchronisation hinzufügen kann.

A.F.7 **Komplette Tabellen werden synchronisiert.**
Alle ausgewählten Tabellen müssen komplett auf allen verbundenen Servern und Clients des Users gespeichert werden.

A.F.8 **Verwaltung von eigenen Tabellen**
Die Bibliothek muss vollständig alle eigenen Tabellen anlegen und verwalten.

A.F.9 **Automatische Verbindung mit Server**
Die Bibliothek muss sich komplett um die Socket Verbindung zum Server kümmern.

A.F.10 **Tabellenstruktur muss nicht angegeben werden.**
Die Bibliothek kann die Struktur der Tabelle selbstständig erkennen.

A.F.11 **Synchronisationsfehler**
Sollten zwei Geräte die gleiche Zeile verändern, muss der Client einen Fehler an die Anwendung werfen, damit diese den Fehler beheben kann.

A.F.12 **Änderungen erkennen durch Zeitstempel**
Änderungen an einer Zeile sollen automatisch mit einem Zeitstempel, der von der Anwendung geführt werden muss, von der Bibliothek erkannt und aufgelöst werden.

2.3 Nicht-funktionale Anforderungen

A.NF.1 **Plattformunabhängigkeit**
Das System muss plattformunabhängig sein.

A.NF.2 **Performance**
Die Synchronisation soll schlank und effizient sein.

A.NF.3 **Gute Nutzbarkeit**
Die Bibliothek soll gut in anderen Projekten nutzbar sein.

A.NF.4 **Keine Veränderung zu klassischer Lösung**
Der Nutzer bekommt keine Veränderung zu einer klassischen Lösung mit, die nur aus einer Datenbank besteht.

A.NF.5 **Android**
Die Bibliothek soll auf Android laufen.

A.NF.6 **Hinzufügen eine Datenbank**
Es sollte möglich sein, eine weitere Datenbanksoftware hinzuzufügen.

2.4 Nichtanforderungen

NA.1 **Aktivitätsträger**
Pro Synchronisierungseinheit gibt es nur einen Aktivitätsträger.

Kapitel 3

Entwurf

In diesem Kapitel werden die Anforderungen in ein Konzept umgewandelt. Dabei wird oft das Akronym MDBS vorkommen. Das ist der Name des eigentlichen Projekts, er steht für ***m**obile **d**ata**b**ase **s**ynchronisation*.

3.1 Grundsätzlicher Aufbau

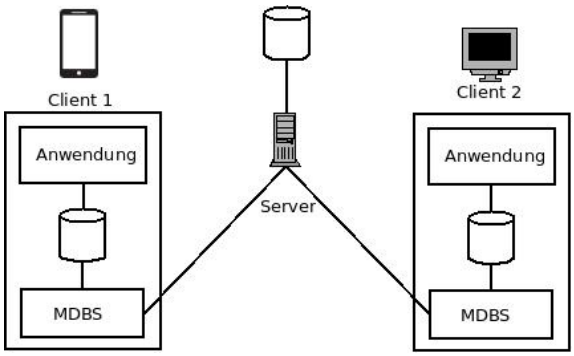

Abbildung 3.1: Grundsätzlicher Aufbau

Die Abbildung 3.1 zeigt den grundsätzlichen Aufbau des Projekts - mit zwei Clients, die mit einem Server verbunden sind. Jeder Teilnehmer hat eine eigene Datenbank, mit der eigenen Kopie der Daten, auf die er immer zugreifen kann. Über das Netzwerk werden die Daten synchronisiert. Somit ist der Datensatz verteilt und das CAP-Theorem (siehe 1.3.2) kommt zum Tragen. Da die Daten immer verfügbar sind, auch wenn das Netz in Partitionen zerfällt, muss es also zu Problemen bei der Konsistenz kommen. Das System unterliegt somit dem Modell der *Eventual consistency* (siehe 1.3.3).

3.2 Aufbau im Detail

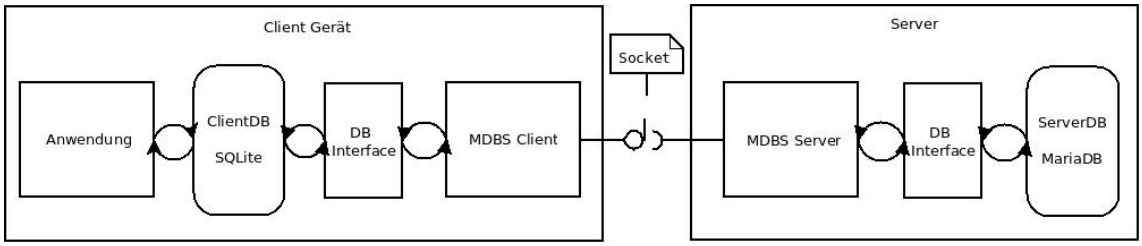

Abbildung 3.2: FMC Schema des Projekts

Die Abbildung 3.2 zeigt den Aufbau des Projekts in der FMC-Notation zwischen einem Client und einem Server. Angefangen von links speichert die Nutzer-Anwendung ihre Daten in der Client Datenbank. Da diese Datenbank auf einem eingebetteten System, zum Beispiel einem Smartphone, funktionieren soll, kommt hier eine SQLite Datenbank zum Einsatz. Der eigentliche MDBS-Client trägt die Sorge, die Daten von der Client-Datenbank auf den Server zu replizieren. Er gelangt an die Datenbank über ein Datenbank-Interface, das für jede Datenbank implementiert werden kann. Auf der Server-Seite nimmt der MDBS-Server die Daten entgegen und speichert diese - auch über ein Datenbank-Interface - in der Server-Datenbank, wofür in diesem Beispiel die Software MariaDB zum Einsatz kommt.

3.3 Bibliotheksschnittstelle

Die Client-Bibliothek bietet der Anwendung eine API, um mit ihr zu kommunizieren. Diese API hat verschiedene Funktionen, um die Anforderungen zu erfüllen.

Zuallererst muss der MDBS-Client erstellt werden. Dafür wird der Konstruktor *public MDBSConnection(DBInterfaceMDBS db)* aufgerufen. Als Parameter gibt man ihm eine passende Implementierung des Datenbank-Interfaces, die mit einer entsprechenden Datenbank bestückt ist. Das Interface und eine Implementierung jeweils für MariaDB und SQLite befinden sich im Paket *eu.t5r.MDBS.sql*.

Als nächstes kann mit der Methode *public void addSyncTable(String tableName, int priority)* eine in der Datenbank existierende Tabelle als synchronisierbar markiert werden. Damit ist die Anforderung *Tabellenauswahl* erfüllt. Um die Vorbereitungen abzuschließen, sollte mit der Methode *public void addSyncListener(SyncListener listen)* ein Event-

listener registriert werden. Dabei muss eine Implementierung des Interface SyncListener übergeben werden. Dieses Interface befindet sich im Paket *eu.t5r.MDBS.structs*. Da die Bibliothek bei der Synchronisation nicht blockiert und auch der Server eine Synchronisation auslösen kann, werden am Ende des Datenaustauschs alle registrierten Listener aufgerufen.

Um nun eine Verbindung mit dem Server aufzubauen, kann über die Methoden *public void setHost(String host)* und *public void setPort(int port)* die IP und der Port des Servers angegeben werden. Über *public void connect()* kann der Client angewiesen werden, eine Verbindung aufzubauen. Da der Server mit einem Benutzernamen und Passwort vor fremden Zugriffen geschützt ist, muss sich der Nutzer vor der ersten Synchronisation mit der Methode *public void logIn(String user, String pwd)* anmelden.

Nun ist alles bereit für das erste Abgleichen der Daten.

Um den Client auf den selben Stand wie der Server zu bringen, muss die Methode *public void sync()* aufgerufen werden. Das sollte am Anfang oder nach jeder Änderung in der Datenbank geschehen.

Am Ende kann die Verbindung mit der Methode *public void close()* geschlossen werden.

3.4 Tabellen-Schema

Das ER-Diagramm 3.3 zeigt das Tabellen-Schema der Datenbanken. Dabei ist zu beachten, dass die DataTable nur ein Beispiel ist, wie eine solche Tabelle aussehen kann. Sie braucht unbedingt eine id und einen timestamp. Welche weiteren Spalten die Tabelle hat, ist dem Programmierer der Anwendung überlassen. Auch kann es in der Datenbank beliebig viele DataTables geben.

Um die Veränderungen in den Datentabellen festzustellen, gibt es eine Meta-Tabelle. In ihr werden für jede Zeile der DataTables die Metadaten geführt: die id in Form eines eindeutigen Identifikators, der Tabellenname, der timestamp als Zeitstempel, der lastTimestamp als letzter Zeitstempel, und ein Boolean flag deleted, um sich alle gelöschten Einträge zu merken. Zwischen den DataTabels und der MetaTabel wird keine Fremdschlüsselbeziehung eingeführt, denn in der MetaTabel werden auch schon gelöschte Zeilen gespeichert und der Nutzer soll diese Tabelle nie zu Gesicht bekommen.

Zuletzt gibt es noch die Commit-Tabelle. In ihr merkt sich der Client und Server alle Tabellen, die er synchronisieren soll und den letzten Commit dieser Tabelle, der für die

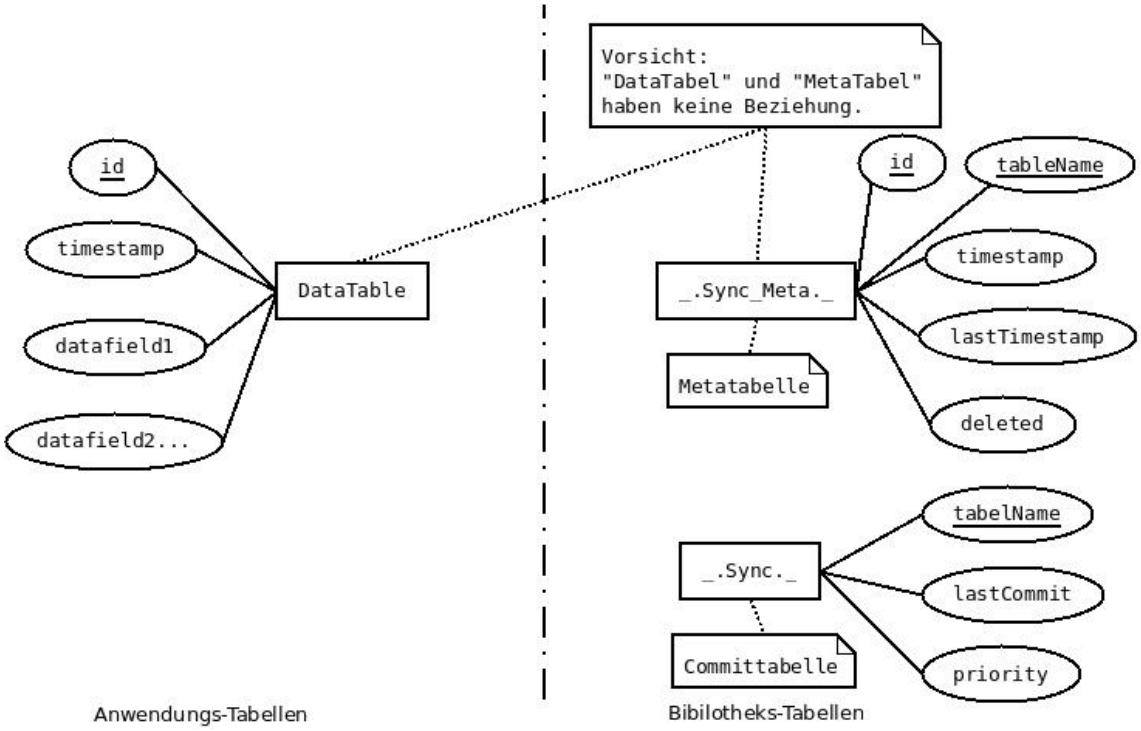

Abbildung 3.3: Entity-Relationship-Diagramm des Projekts

inkrementelle Synchronisierung gebraucht wird. Um eine Reihenfolge, in der die Änderungen in der Tabelle gemacht werden, festzulegen, gibt es für jede Tabelle noch eine Priorität. Desto kleiner die Zahl, desto früher wird die Veränderung vorgenommen. Diese Funktion ermöglicht zum Beispiel Fremdschlüssel-Beziehungen zu benutzen.

3.5 Eindeutige Identifizierung

Wie im vorigen Kapitel 3.4 schon angesprochen, wird eine *id* und ein *timestamp* als eindeutige Identifizierung genutzt. Was in diesen beiden Zeilen benutzt wird, wird zwar an keiner Stelle geprüft, aber es wird strengstens empfohlen, eine UUID und einen Zeitstempel, der bei jeder Änderung auf die aktuelle Zeit geändert wird, zu benutzen. Dabei kann sich in beiden Fällen auf die Java-Implementierung verlassen werden. Für die UUIDs sollte der Befehl *UUID.randomUUID().toString()* aus dem entsprechenden Paket aufgerufen werden und für die Zeitstempel kann der Befel *System.currentTimeMillis()* benutzt werden. Da die Funktion *currentTimeMillis()* die Millisekunden seit dem 1. Januar 1970 UTC zählt, kommt es nicht zu Problemen bei verschiedenen Zeitzonen. Ein *Universally Unique Identifier*, kurz UUID, ist ist eine 128 bit lange Kennung, die garantiert, dass sie über Zeit und Raum einzigartig ist [PL05].

3.6 Synchronisierung

Die Synchronisierung der Datenbanken ist das Hauptelement des Projekts. Sie soll eine 1 : 1 Replikation zwischen Server und Client schaffen. Dabei kann sowohl der Client als auch der Server diesen Prozess initiieren.

Um möglichst wenig Bandbreite zu nutzen, gibt es zwei Arten von Synchronisation.

3.6.1 Voll

Bei der vollen Synchronisation schickt der Initiator der Synchronisation seine kompletten Meta-Daten an sein Gegenstück. Der Empfänger kann nun die empfangenen Meta-Daten mit seinen Meta-Daten vergleichen und die Unterschiede in Form von *inserts*, *updates* und *deletes* in beide Richtungen feststellen. Mit diesem Wissen kann er folgendes zurückschicken: die Daten, die auf seiner Seite neuer sind, eine Anfrage der Daten die auf der Gegenseite neuer sind und einen neuen Commit. Der Initiator kann nun die Änderungen in seine Datenbank einpflegen und die Datenanfragen beantworten, die dann auch auf seiner Gegenseite eingepflegt werden.

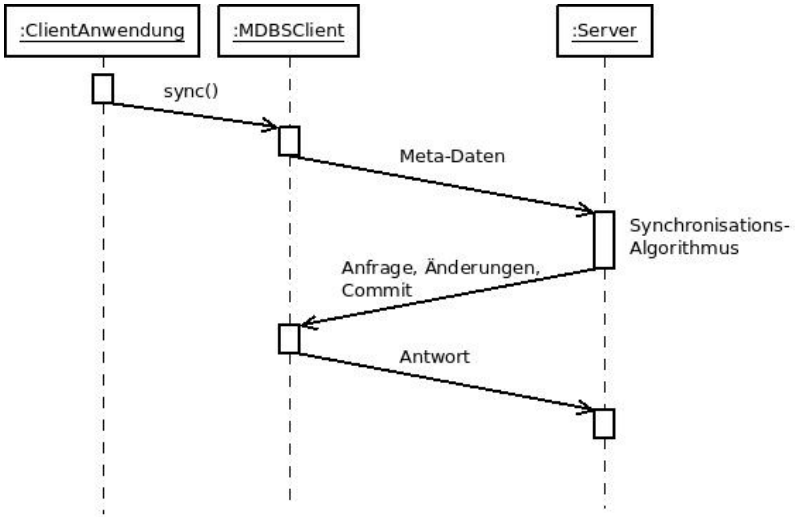

Abbildung 3.4: Sequenzdiagramm der Voll-Synchronisation

Dieser Ablauf (Abbildung 3.4) lehnt sich an den Three-Way-Handshake an.

3.6.2 Inkrementell

Bei der heutigen Internetverbreitung, auch auf Mobilgeräten, kann man davon ausgehen, dass die Clients meist mit dem Server verbunden sind. Das bedeutet, die meiste Zeit sind die Datensätze exakt gleich. In Verbindung mit der Prämisse, dass die meisten Änderungen nur einzelne Zeilen betreffen, können diese Differenzen auch inkrementell übertragen werden. Hierbei müssen weniger Daten übertragen werden, allerdings funktioniert der Ansatz nur, wenn die Tabellen auf dem exakt gleichen Stand sind.

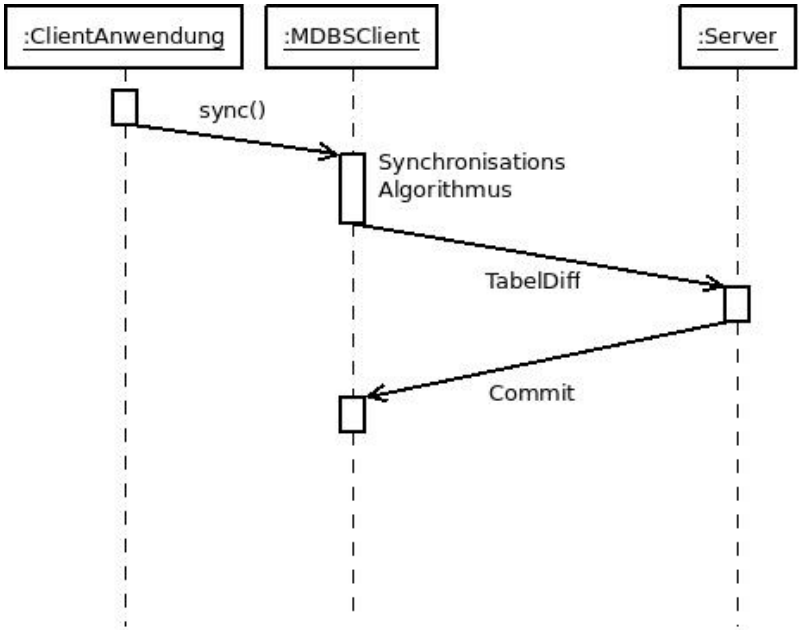

Abbildung 3.5: Sequenzdiagramm der Inkrementellen-Synchronisation

Das Sequenzdiagramm 3.5 zeigt den Ablauf. Die Anwendung stößt nach einer Änderung der Daten die Synchronisation an. Die Tabellen werden mit der Meta-Tabelle verglichen. Sollten Zeilen noch nicht in der Meta-Tabelle stehen, sind sie hinzugekommen. Stehen Zeilen nicht mehr in der Datentabelle, wurden sie gelöscht. Haben sich die Zeitstempel der Zeilen geändert, müssen auch diese Zeilen synchronisiert werden. Alle Änderungen werden gesammelt und zum einen - mit den Daten gespickt - an die Gegenstelle verschickt und zum anderen in die Meta-Tabelle geschrieben, um die internen Tabellen wieder auf den gleichen Stand zu bekommen. Die Gegenstelle führt die geschickten Änderungen in ihren Tabellen aus und erstellt einen neuen Namen für den Commit. Den verschickt sie als Bestätigung an den Sender.

3.7 Socket und Serialisierung

Zur Kommunikation zwischen dem Client und dem Server wird der in Java vorhandene TCP/IP Socket benutzt.

Die Daten werden mit Hilfe des *java.io.ObjectInputStream* beziehungsweise *java.io.ObjectOutputStream* serialisiert. Mit diesen beiden Klassen können Objekte, deren Klassen das Serializable Interface implementieren, ohne zusätzlichen Programmieraufwand serialisiert werden. Dabei wird immer eine Containerklasse, in diesem Projekt *SocketContainer* genannt, gesendet. Diese enthält ein Objekt, das die eigentlichen Daten beinhaltet, und den Typ des Objekts, um es richtig casten zu können.

3.8 Prioritäten

Moderne Datenbanken besitzen Fremdschlüsselbeziehungen, um Tabellen miteinander zu verbinden. Dabei muss das Tabellenattribut, auf das verwiesen werden soll, zuerst vorhanden sein. Somit muss bei einer Synchronisierung darauf geachtet werden, dass eine Reihenfolge eingehalten wird. Diese Folge wird durch eine Priorität, die beim Aktivieren im MDBS angegeben wird, festgelegt. Dabei werden die Änderungen mit aufsteigender Priorität durchgeführt. Die Änderungen, die zu einer Tabelle mit einer kleinen Priorität gehören, werden zuerst ausgeführt. Tabellen, die mit einem Fremdschlüssel auf eine andere Tabelle verweisen, sollten somit eine höhere Priorität als die referenzierte Tabelle bekommen, um später bei der Synchronisierung an der Reihe zu sein.

3.9 Datenbankinterface

Das MDBS ist nicht auf ein paar bestimmte Datenbanksysteme festgelegt. Es soll mit vielen verschiedenen Datenhaltungssystemen zurechtkommen. Um das zu ermöglichen, benutzt das Projekt ein Interface, um an seine Daten zu gelangen. Dieses Interface muss für jedes Datenhaltungssystem implementiert werden. Für SQLite und MariaDB wurde schon eine Implementierung geschrieben. Grundsätzlich kann für jede relationale Datenbank eine Implementierung geschrieben werden. Es können aber auch ähnliche Datenbanken genutzt werden, zum Beispiel Apache Cassandra, hier muss aber eine tabellenähnliche Struktur vorliegen und bis auf die Konsistenz das ACID-Theorem eingehalten werden.

3.10 Authentifizierung

Um den Datenbestand vor unautorisiertem Zugriff zu schützen, muss sich jeder Client zuerst authentifizieren, um danach vom Server autorisiert zu werden. Ohne diesen Prozess wird keine Synchronisierung zugelassen.

3.11 Event handler

Ruft die Anwendung das MDBS zum Synchronisieren auf, blockiert der Aufruf nicht während der Synchronisation. Kommt der Aufruf also zurück, ist die Synchronisierung noch nicht abgeschlossen. Nach dem Datenaustausch haben sich aber womöglich die Daten geändert. Somit sollte die Anwendung benachrichtigt werden, damit sie dem Benutzer die neuen Werte anzeigen kann. Dies wird mithilfe eines Event-Handlers bewerkstelligt. Die Anwendung übergibt dem MDBS eine Implementierung des Interfaces *SyncListener*, das nur die Methode *syncPerformed* besitzt. Alle SyncListener werden nach einer erfolgreichen Synchronisation aufgerufen. Somit können die Anwendungen reagieren.

Kapitel 4

Implementierung

4.1 Code Aufbau

Das Projekt ist in drei Teilprojekte aufgeteilt. Der Client befindet sich im Ordner *MDBS/*, der Server im Ordner *MDBSServer/* und das Testprojekt im Ordner *MDBSTest/*. Das Testprojekt ist recht klein. Es enthält im Testordner zwei JUnit-Klassen, die die Funktionen der Projekte einmal mit SQLite und einmal mit MariaDB testen. Dafür wird der Server und der Client als komplettes Projekt importiert.

Die folgenden Pakete werden mit einer Ordnerverknüpfung, sowohl im Client als auch im Server, benutzt:

- *eu.t5r.MDBS.sql*: enthält das Datenbankinterface und zwei Implementierungen
- *eu.t5r.MDBS.structs*: enthält alle Grundstrukturen um Daten darzustellen
- *eu.t5r.MDBS.structs.socket*: enthält alle Strukturen, die Daten über das Netzwerk übertragen
- *eu.t5r.MDBS.structs.types*: enthält Enumerationen
- *eu.t5r.MDBS.sync*: enthält Algorithmen

Übrig bleiben der Socketserver und -client im Paket *eu.t5r.MDBSServer.socket* und *eu.t5r.MDBS.socket*, sowie die Hauptdateien im Paket *eu.t5r.MDBSServer* und *eu.t5r.MDBS*

Im Ordner *example/* befinden sich zwei beispielhafte Implementierungen.

4.2 Synchronisationsalgorithmus

Das Herzstück ist der Synchronisationsalgorithmus. Genau genommen gibt es zwei Funktionen, die diese Aufgabe übernehmen: *checkDiffOneWay()* findet die Änderungen in der Datentabelle relativ zur Meta-Tabelle und *checkDiffBothWay* findet die Änderungen zwischen einem Client und einem Server. Dabei kann der erste Algorithmus nur die Veränderungen in eine Richtung finden und der Zweite in beide Richtungen, daher auch die Namen. Da das auch der Hauptunterschied ist, schauen wir uns nur den ersten Algorithmus an:

```
1    public static List<GenDiffContainer<RowContainer>> checkDiffOneWay(List<MetaData>
         a, List<MetaData> b) {
2        List<GenDiffContainer<RowContainer>> result = new ArrayList<>();
```

Die öffentliche statische Methode, die in der Klasse *Algorithm* zu finden ist, nimmt zwei Listen von Meta-Daten und gibt eine Liste von Generischen Diff-Containern zurück. Diese Ergebnisliste wird direkt zu Anfang erstellt.

```
1        meta_for:
2        for (MetaData aa : a) {
3            for (MetaData bb : b) {
4                if (aa.equalsUUID(bb)) {
```

Nachdem eine Sprungmarke gesetzt wurde, wird jede Meta-Information der einen Liste mit jeder Meta-Information der anderen Liste verglichen, um Zeilen mit dem gleichen Identifier zu finden.

```
1                    if (aa.getSynctime() > bb.getSynctime()) {
2                        // found update
3                        /// uuid, new time from a, old synctime from b
4                        result.add(new GenDiffContainer(DiffTypes.UPDATE,
                             DirectionTypes.SERVER_TO_CLIENT, new
                             RowContainer(aa.getTableName(), aa.getUuid(),
                             aa.getSynctime(), bb.getSynctime(), false)));
5                    }
```

Sollte der Zeitstempel der Zeile in der Datentabelle neuer sein, speichert sich die Funktion die Änderung.

```
1                    continue meta_for;
2                }
```

```
3             }
```

In jedem Fall wurde ein Zeilenpärchen gefunden und es kann das nächste gesucht werden.

```
1             }
2
3             // found insert
4             result.add(new GenDiffContainer(DiffTypes.INSERT,
                  DirectionTypes.SERVER_TO_CLIENT, new RowContainer(aa.getTableName(),
                  aa.getUuid(), aa.getSynctime(), 0, false)));
5
6         }
```

Sollte kein passendes Gegenstück gefunden werden, muss die Zeile neu sein und auch das wird gespeichert.

```
1         meta_del_for:
2         for (MetaData bb : b) {
3             for (MetaData aa : a) {
4                 if (bb.equalsUUID(aa)) {
5                     continue meta_del_for;
6                 }
7             }
8             // found delete
9             result.add(new GenDiffContainer(DiffTypes.DELETE,
                  DirectionTypes.SERVER_TO_CLIENT, new RowContainer(bb.getTableName(),
                  bb.getUuid(), 0, 0, true)));
10        }
```

Sollte eine Zeile gelöscht werden, fehlt diese in der zweiten Liste, deshalb müssen die beiden Listen noch einmal in umgekehrter Reihenfolge verglichen werden, um die gelöschten Zeilen zu finden.

```
1         return result;
2     }
```

Zum Schluss werden die gefundenen Änderungen zurückgegeben und die Funktion beendet.

4.3 getData-Methode

Die getData-Methode ist eine weitere wichtige und interessante Methode. Sie befindet sich in der SQLite Implementierung des Datenbankinterfaces und kann die Daten einer

beliebigen Zeile lesen. Dabei erfüllt sie die Anforderung *Tabellenstruktur muss nicht angegeben werden* und erkennt die abgelegten Daten selbstständig.

```
1    public void getData(RowContainer result) throws SQLException {
```

Die Methode bekommt einen *RowContainer*, der laut Spezifizierung nur mit den *MetaDataDeep* gefüllt sein darf.

```
1        String sql = "SELECT * FROM '" + result.getTableName() + "' WHERE id = ?;";
2        PreparedStatement ps = self.prepareStatement(sql);
3        ps.setString(1, result.getUuid());
4
5        ResultSet rs = ps.executeQuery();
```

Zu Beginn wird die SQL-Abfrage vorbereitet und ausgeführt. Sie fragt die Zeile ab, die der Container vorgibt.

```
1            ResultSetMetaData meta = rs.getMetaData();
2            for (int i = 1; i <= meta.getColumnCount(); i++) {
3
4                MDBSTypes type;
5                switch (meta.getColumnTypeName(i)) {
6                    case "TEXT":
7                        type = MDBSTypes.TEXT;
8                        break;
9                    case "INTEGER":
10                       type = MDBSTypes.INTEGER;
11                       break;
12                   case "REAL":
13                       type = MDBSTypes.REAL;
14                       break;
15                   case "BLOB":
16                       type = MDBSTypes.BLOB;
17                       break;
18
19                   default:
20                       throw new SQLException("Not Defined Type: " +
                             meta.getColumnTypeName(i));
21               }
```

Nun wird für jede Zelle der Typ herausgefunden.

```
1                result.addCell(meta.getColumnName(i), new DataContainer(type,
                     rs.getObject(i)));
```

Zu guter Letzt wird die Zelle mit ihrem Typ als Objekt im Container gespeichert.

4.4 Android

Die Implementierung und Portierung auf Android erwies sich als schwieriger als angenommen. Deshalb hier einige Informationen dazu:

Erstens unterstützen Android erst ab der Version 7.0 einige Java 8 Features und auch nicht Alle [and17]. Wie im Graph 4.1 zu sehen ist, macht Android 7 nur 10% des Marktanteils aus. Deshalb musste für die MDBS-Client-Bibilothek die Java Version 7 benutzt werden.

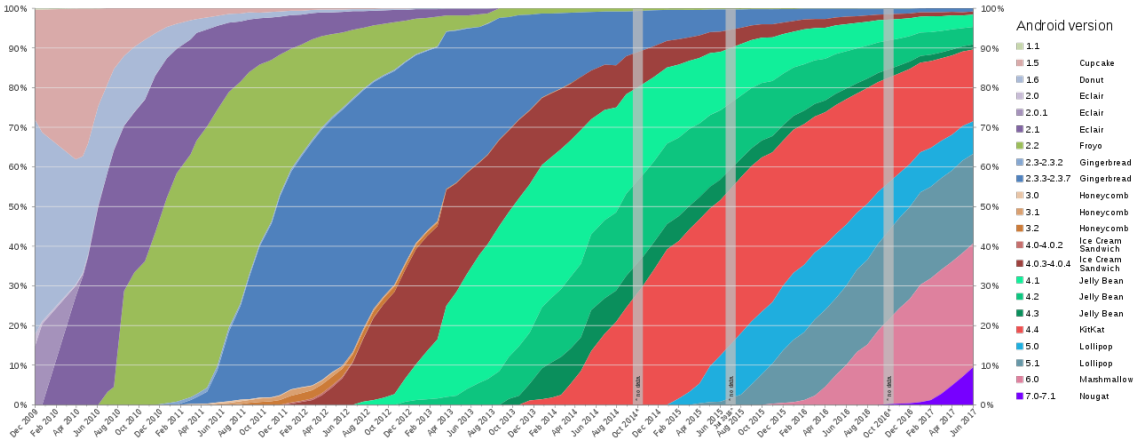

Abbildung 4.1: Häufigkeit der verschiedenen Android-Versionen. Alle Versionen älter als 4.0 sind fast verschwunden.[Com17]

Um auf Android eine SQLite Datenbank zu nutzen, kann ein Entwickler nicht einfach den Standard JDBS-Driver verwenden, da dieser nicht für Android gedacht ist. Deshalb bietet Android hier eine eigene Implementierung mit dem Namen *android.database.sqlite*, die ein wenig andere Methoden hat. Um die MDBS-Bibilothek nun auf Android zum Laufen zu bekommen, wurde für diese Datenbank auch das DBInterface implementiert.

Zum Schluss ist aufgefallen, dass die *ObjectStream*s nicht versionskompatibel zu sein scheinen. Da der Client, wegen Android, auf Java-Version 7 war und der Server auf Version 8, kam es zu einer *Broken pipe* Exception. Deshalb wurde kurzerhand auch der Server auf die Java-Version 7 umgestellt.

Kapitel 5

Tests und Evaluation

Um das Projekt zu testen, sowohl in Funktion als auch in Benutzbarkeit, wurden JUnit-Tests und eine Beispielanwendnung geschrieben.

5.1 JUnit

Mit JUnit wurde die Funktion der Bibliothek in Zusammenspiel mit dem Server getestet. Da JUnit keine Socket- und Thread-Test unterstützt, musste das Projekt etwas untypisch getestet werden. Vor jedem Test werden die alten Datenbanken entweder gelöscht oder geleert. Danach werden drei Datenbanken erstellt und auf einen Server und zwei Clients verteilt. Nun kann das eigentliche Testen beginnen. In den ersten Tests werden noch sehr typisch Methoden abgefragt und getestet, ob die richtigen Rückgabewerte erstellt werden. Soll allerdings ein Client Daten auf einen Server synchronisieren, muss ein anderer Weg genutzt werden. Dazu wird ein neuer Wert in die Datenbank der Clients geschrieben und die Synchronisation angestoßen. Nach einer Wartezeit von einer halben Sekunde wird auf dem Server mittels JUnits *asserts* geprüft, ob die erwarteten Daten angekommen sind. Mit dieser Technik können auch mehrere Aktivitätsträger getestet werden. Allerdings entsteht durch diese Testart eine Abhängigkeit der Tests untereinander. Deshalb müssen die Tests in einer Reihenfolge ausgeführt werden. Dies wird mittels der Annotation $@FixMethodOrder(MethodSorters.NAME\_ASCENDING)$ erreicht.

Folgende Teile wurden getestet:

A **Vorbereitung** Tabelle Person erstellt

B **Hinzufügen** Tabelle Person im MDBS hinzugefügt

C **Überprüfen** Meta-Tabellen werden erstellt

D **Gleichheit** Hier wird eine equals Methode getestet.

E **Synchronisation** Die beiden Synchronisations-Algorithmen und das Schreiben der Änderungen in die Meta-Tabelle wird getestet

F **Server** Der Client baut eine Verbindung zum Server auf und meldet sich an

G **Socket-Synchronisation** Alle drei Datenbank-Operationen (insert, update, delete) werden zwischen Server und Client getestet

H **Mehrere Tabellen** Es wird getestet, ob auch mehrere Tabellen voneinander getrennt synchronisiert werden können

I **Inkrementell** Das inkrementelle Synchronisieren wird eingeschaltet und wie die normale Socket-Synchronisation getestet

J **Mehrere Geräte** Ein zweiter Client wird mit dem Server verbunden und es wird getestet, ob auch hier die Daten ankommen

K **Trigger** Ein Trigger wird dem zweiten Client übergeben und getestet

L **Fremdschlüssel** Es wird getestet, ob die Tabellen - der Priorität nach sortiert - in der richtigen Reihenfolge bedient werden

In einigen Testfällen wird kein direkter Zustand getestet, sondern andere Fälle vorbereitet. Hier darf aber auch keine Exception entstehen.

Zusätzlich wurden alle aufgeführten Tests einmal für SQLite und einmal für MariaDB getestet.

5.2 Beispiel-Anwendung

Um zu testen, ob sich die Bibliothek auch gut in eine Anwendung einbauen lässt, wurde die schon mehrfach erwähnte Notiz-App umgesetzt. Dafür wurde eine einfache java-swing GUI gebaut und der Server dafür vorbereitet.

Wie in der Abbildung 5.1 zu sehen ist, unterstützt die Anwendung das Verbinden und Anmelden am Server. Notizen können mit einem Titel und Text angelegt und gelöscht werden. Bei jeder Änderung wird automatisch eine Synchronisation angestoßen und die

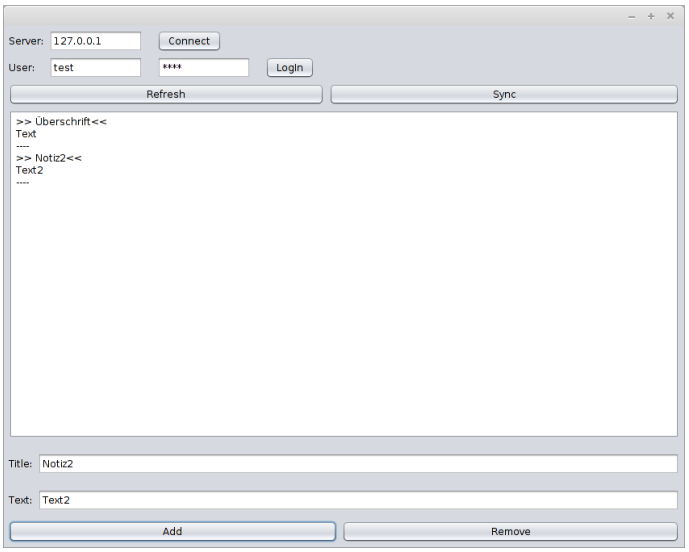

Abbildung 5.1: Desktop-GUI

große Textbox aktualisiert. Auch wenn eine Änderung vom Server kommt, wird über ein Trigger alles erneuert. Zu Testzwecken kann der Nutzer die Synchronisation der Datenbank und Aktualisierung der GUI mit zwei Buttons von Hand anstoßen.

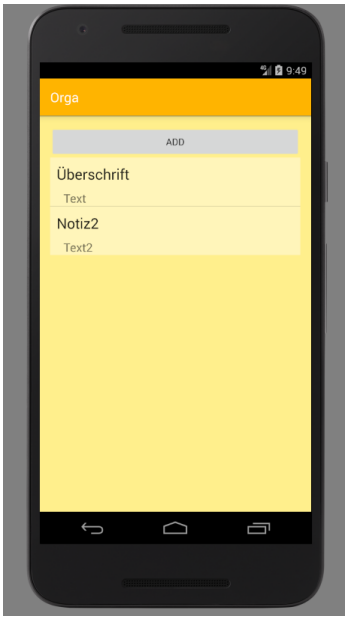

Abbildung 5.2: Android-GUI

Für Android (siehe Abbildung 5.2) wurde eine äquivalente Anwendung geschrieben. In ihr wird die Verbindung und Anmeldung am Server allerdings automatisch gemacht.

5.3 Vergleichbare Arbeiten

Sucht man im Internet nach Datenbank Synchronisierung für mobile Anwendungen, zeigt die Suchmaschine entweder Tutorials zum selber bauen oder das Projekt *Couchbase Lite* an.

Couchbase Lite gibt es in verschiedenen Implementierungen. Sie bietet ähnliche Funktionalitäten wie dieses Projekt. Allerdings wird hier eine API benutzt, um auf die Datenbank zuzugreifen und es wird auf dem Server ein eigenes Datenbanksystem benutzt, den Couchbase Server.

Kapitel 6

Fazit

6.1 Ergebnisse

Es ist ein System entwickelt und gebaut worden, dass zwei verschiedene Datenbanksysteme über ein nicht immer verfügbares Netzwerk synchronisieren kann (A.F.1). Dabei werden die Daten komplett von den Clients über den Server automatisch verteilt und gespeichert (A.F.7, A.F.3 , A.F.9 u. A.F.2). Die Daten können zu jeder Zeit und von jedem Client gelesen und geändert werden (A.F.4 u. A.F.5). Tabellen die synchronisiert werden sollen, können dem System übergeben werden (A.F.6). Die Bibliothek erkennt die Struktur der Daten (A.F.10) und legt alleine innere Strukturen (A.F.8) an, um Änderungen zu erkennen (A.F.12). Dabei stellt nur die Bibliothek eine Socketverbindung mit dem Server her (A.F.9). Das System ist aber nur für kleine personenbezogene Daten geeignet, da immer die komplette Tabelle übertragen wird (Vorbedingung).

Durch die Nutzung von Java kann die Software auf allen Betriebssystemen laufen, für die es eine JVM gibt (A.NF.1) - auch auf Android (A.NF.6). Zwar wurden keine Massen- oder Performancetests durchgeführt (A.FN.2), aber in Unit-Tests und einem Anwendungsbeispiel konnte gezeigt werden, dass die Bibliothek gut zu nutzen ist (A.FN.3). Des Weiteren greift die Anwendung direkt auf die Datenbank zu, ohne den Umweg über die Bibliothek zu nehmen (A.FN.4). Zusätzlich ist es möglich weitere Datenbanksysteme hinzuzufügen (A.FN.6).

6.2 Ausblick

Das Projekt soll nicht nur als Grundlage einer Bachelor-Thesis dienen, sondern soll auch als freie Software veröffentlicht werden. Zudem soll die Bibliothek auch benutzt werden, um Anwendungen und Apps zu entwickeln.

6.3 Offenes

Leider sind bei der Entwicklung des Prototypen einige Dinge liegen geblieben, die zur vollständigen Benutzung noch fehlen.

Serialisierung über JSON

Um später unabhängiger von Java zu sein, soll nicht mehr die Java-Serialisierung benutzt werden, sondern JSON. Damit können auch Implementierungen in anderen Sprachen programmiert werden. Zudem können die Daten über einen Websocket gesendet werden, um nicht von Portsperren behindert zu werden.

Nutzerdaten trennen

Leider ist bei der Anforderungsanalyse eine intuitive Anforderung durch das Raster gefallen. Somit wurde in der Implementierung vergessen, die Nutzerdaten zu trennen. Zurzeit ist die Anmeldung lediglich dazu da, um unautorisierte Zugriffe abzufangen. In der fertigen Implementierung soll aber jeder Nutzer seine eigenen Daten besitzen, die niemand anderes einsehen kann.

Fehlerbehandlung

Die Fehlerbehandlung des Projekts ist noch nicht zufriedenstellend. Fehler, die auf dem Server passieren, aber den Client betreffen, werden noch nicht weitergegeben. Zudem muss bei bestimmten Fehlern, etwa wenn eine Zeile von zwei Clients gleichzeitig verändert wird, eine besondere Fehlerbehandlung gemacht werden.

Falsche Anfragen abfangen

Zur Sicherheit sollen in Zukunft falsche oder unautorisierte Anfragen herausgefiltert werden.

Verschlüsselung

Die Daten sollen in der finalen Version verschlüsselt übertragen werden.

Löschen obsoleter Meta-Daten

Der Server und auch der Client müssen sich auch die Meta-Daten von schon gelöschten Zeilen merken, damit sich gelöschte Zeilen nicht immer wieder als neue Zeilen von anderen Geräten in das System einschleichen. Allerdings könnten diese Daten sehr groß werden. Hier muss eine Lösung entwickelt werden, die die Daten nach einer gewissen Zeit aufräumt.

Mehrere Anwendungen pro Server

Um einen Server besser auszulasten, soll es möglich sein, mehrere Anwendungen auf ihm abzulegen.

Kapitel 7

Literaturverzeichnis

[and17] ANDROID.COM: *Use Java 8 language features*. https://developer.android.com/guide/platform/j8-jack.html. Version: 2017. – [Online; Stand 30. September 2017]

[Com17] COMMONS, Wikimedia: *File:Android historical version distribution - vector.svg — Wikimedia Commons, the free media repository*. https://commons.wikimedia.org/w/index.php?title=File:Android_historical_version_distribution_-_vector.svg&oldid=250291417. Version: 2017. – [Online; accessed 30-September-2017]

[dig14] DIGITALOCEAN: *SQLite vs MySQL vs PostgreSQL: A Comparison Of Relational Database Management Systems*. https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-databa
Version: 2014. – [Online; Stand 8. August 2017]

[eng17] ENGINES.COM db: *DB-Engines Ranking - die Rangliste der populärsten Datenbankmanagementsysteme*. https://db-engines.com/de/ranking. Version: 2017. – [Online; Stand 7. August 2017]

[Lab14] LABS, Flux7: *CAP Theorem: Its importance in distributed systems*. http://blog.flux7.com/blogs/nosql/cap-theorem-why-does-it-matter. Version: 2014. – [Online; Stand 10. August 2017]

[Mar17] MARIADB: *MariaDB versus MySQL – Kompatibilität - MariaDB Knowledge Base*. https://mariadb.com/kb/de/mariadb-vs-mysql-compatibility/. Version: 2017. – [Online; Stand 7. August 2017]

[PL05] P. LEACH, R. S. M. Mealling M. M. Mealling: A Universally Unique IDentifier (UUID) URN Namespace. 2005 (4122). – RFC. – 1–30 S.

[sql17] SQLITE.ORG: *Datatypes In SQLite Version 3*. https://sqlite.org/datatype3.html. Version: 2017. – [Online; Stand 15. August 2017]

[VB17] VOLKER BRIEGLEB heise.de: *Smartphone-Markt: Windows spielt keine Rolle mehr*. https://heise.de/-3722875. Version: 2017. – [Online; Stand 08. Oktober 2017]

[Vos] VOSSEN, Gottfried: *Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme*. OLDENBOURG WISSENSCHAFTSVERLAG

[Was16] WASZLAVIK, Márton: *Demystifying CAP theorem, Eventual Consistency and exactly once delivery guarantee*. https://medium.com/@marton.waszlavik/demystifying-cap-theorem-eventual-consistency-and-exactly-once-d Version: 2016. – [Online; Stand 10. August 2017]

[Wik16] WIKIPEDIA: *CAP-Theorem — Wikipedia, Die freie Enzyklopädie*. https://de.wikipedia.org/w/index.php?title=CAP-Theorem&oldid=160605531. Version: 2016. – [Online; Stand 10. August 2017]

[Wik17a] WIKIPEDIA: *ACID — Wikipedia, Die freie Enzyklopädie*. https://de.wikipedia.org/w/index.php?title=ACID&oldid=165251620. Version: 2017. – [Online; Stand 10. August 2017]

[Wik17b] WIKIPEDIA: *Android (Betriebssystem) — Wikipedia, Die freie Enzyklopädie*. https://de.wikipedia.org/w/index.php?title=Android_(Betriebssystem)&oldid=167995762. Version: 2017. – [Online; Stand 15. August 2017]

[Wik17c] WIKIPEDIA: *Konsistenz (Datenspeicherung) — Wikipedia, Die freie Enzyklopädie*. https://de.wikipedia.org/w/index.php?

title=Konsistenz_(Datenspeicherung)&oldid=163828232. Version: 2017. – [Online; Stand 10. August 2017]

[Wik17d] WIKIPEDIA: *MySQL — Wikipedia, Die freie Enzyklopädie*. https://de.wikipedia.org/w/index.php?title=MySQL&oldid=167166180. Version: 2017. – [Online; Stand 8. August 2017]

[Wik17e] WIKIPEDIA: *SQLite — Wikipedia, Die freie Enzyklopädie*. https://de.wikipedia.org/w/index.php?title=SQLite&oldid=167924414. Version: 2017. – [Online; Stand 7. August 2017]

Abbildungsverzeichnis

Verzeichnis der Quellcodes