Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Synchronisations- und Versionierungsstrategien für mobile Daten

Admir Obralija

Studiengang:	Softwaretechnik
Prüfer/in:	PD Dr. rer. nat. habil. Holger Schwarz
Betreuer/in:	Dr. rer. nat. Christoph Stach
Beginn am:	13. Dezember 2017
Beendet am:	13. Juli 2018
CR-Nummer:	C.2.2, C.2.4, D.4.6, E.4

Kurzfassung

Mobile Datenbanksysteme gehören mitunter zur Grundlage für die strukturierte Datenhaltung in mobilen Anwendungen. Die Vielfalt und die zunehmende Vernetzung von mobilen Endgeräten führt dazu, dass Datenbestände vermehrt von mehreren Benutzern gemeinsam genutzt und bearbeitet werden. Die hierfür erforderlichen, zentralen Datenbanksysteme, die die gemeinsam genutzten Daten aufbewahren sollten, sind jedoch von mobilen Endgeräten aus nicht direkt anzusprechen. Außerdem sind bestehende Ansätze, die einen Datenabgleich zwischen einer mobilen und einer zentralisierten Datenbank anstreben, nicht für die sich ergebenden Anforderungen ausgelegt und somit für eine kollaborative Datenbearbeitung weitestgehend ungeeignet.

Die vorliegende Arbeit stellt ein verbessertes Konzept für die Synchronisation und den Austausch von mobilen Daten vor, bei der ein Synchronisationsserver die Schnittstelle zwischen den mobilen Clients und der gemeinsam genutzten Datenbank bildet. Es wird ein zustandsbasierendes Synchronisationsprotokoll vorgestellt, welches sowohl client- als auch serverseitig implementiert wird und eine effiziente Synchronisation erreicht. Dabei werden die bei der kollaborativen Bearbeitung von Daten auftretenden Probleme unter Zuhilfenahme einer Versionierungsstrategie präemptiv verhindert. Das Synchronisationsverfahren wird anhand einer prototypischen Implementierung sowohl anwendungsbezogen als auch technisch evaluiert und den vorhergehenden Ansätzen gegenübergestellt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

COMEDA Collective Medical Data Assistant

DBMS Datenbankmanagementsystem

DEA Deterministisch endlicher Automat

ECHO Enhancing Chronic patients' Health Online

IoT Internet of Things

JSON JavaScript Object Notation

MRDMS Mobile Replicated Database Management Synchronization

MVP Model-View-Presenter

PMP Privacy Management Platform

RDBMS Relationales Datenbankmanagementsystem

SAMD Synchronization Algorithms based on Message Digest

SDC Secure Data Container

SQL Structured Query Language

SSPMD State based Synchronization Protocol for Mobile Databases

TLS Transport Layer Security

XML Extensible Markup Language

YAML YAML Ain't Markup Language

1 Einführung

Mobile Endgeräte gewannen in den letzten zwei Jahrzehnten zunehmend an Bedeutung sowohl in privaten als auch in dienstlichen Bereichen des Alltags. Nicht zuletzt die Ortsunabhängigkeit und die damit einhergehende Zeitersparnis, sondern auch die stetig steigende Kompaktheit dieser mittlerweile vollwertigen Rechner verschafften ihnen große Popularität. So konnte im Jahr 2010 ein weltweiter Verkauf von mehr als 250 Millionen Smartphones verzeichnet werden, bei einem Zuwachs von 67% vergleichend mit dem Jahr zuvor [Cro10]. Begleitend erschienen auch mobile Anwendungen (kurz: *Apps*) für verschiedene Anwendungsbereiche; von einfacher Kommunikation bis hin zu komplexen Systemen zur Datenverwaltung [JB12; Ven14].

Der Bezug von Informationen anderer mobiler Geräte sowie dessen lokale Bearbeitung und Aufbewahrung rückten insbesondere durch die Verbreitung des *Internets der Dinge* (engl. *Internet of Things*, IoT) in den Vordergrund [XDC+14]. Im IoT stellen mobile Geräte, die u. a. eine überwachende Tätigkeit haben können, die gewonnenen Daten über das Netzwerk bereit, die dann schließlich für eine resultierende Aktion auf einem anderen Gerät ausgewertet werden.

Für eine strukturierte und persistente Datenhaltung wurde auch für mobile Geräte eine Reihe an *relationalen Datenbankmanagementsystemen* (RDBMS) entwickelt. Einen der häufigen Vertreter für Android-Plattformen stellt das *SQLite*[1] Datenbanksystem dar. Ein RDBMS normiert durch Vorgabe einer Abfragesprache anwendungsübergreifend den Zugriff auf die Datenbestände. Es agiert eigenständig in Form eines Dienstes und kann nicht direkt angesprochen werden, weshalb eine geeignete Bibliothek als Programmierschnittstelle hierfür angeboten wird.

Aufgrund des hohen Anspruchs an Hard- und Software, die ein stationäres Datenbanksystem mit sich bringt, ist dessen Ausführung auf mobilen Geräten bislang nicht möglich. Stattdessen muss auf speicher- und energieschonende Lösungen zurückgegriffen werden [JSY09].

1.1 Problem und Motivation

Die für mobile Plattformen konzipierte RDBMS stellen eine Alternative zur Aufbewahrung lokaler Daten dar. Jedoch steigt mit zunehmender Vernetzung der Bedarf an Möglichkeiten, diese lokal vorhandenen Daten mit anderen Personen oder Geräten im Netzwerk teilen zu können. Ein Beispiel hierfür sind Anwesenheitszeiten der Mitglieder eines Haushaltes, die von unterschiedlichen Smart Devices aus mittels einer App modifiziert werden, um auf diese Weise den Stromverbrauch durch elektrische Verbraucher zu senken.

[1] https://www.sqlite.org/

In jüngster Zeit wurden zahlreiche Konzepte eingeführt, um Daten zwischen mobilen Geräten auszutauschen. Da Apps in vielen technischen und sozialen Bereichen bis hin in den gesundheitlichen Sektor Einzug halten, entwickelt sich eine kollaborative Form, in der die Daten bearbeitet werden. Ferner reicht ein einfacher Datenaustausch nicht länger aus, um diese kollaborative Bearbeitung zu ermöglichen, wodurch zentralisierte und über das Netzwerk erreichbare Datenbanken als ein gemeinsamer Stützpunkt für die anfallenden Daten verwendet werden müssen.

Eine Verbindung zu solchen Datenbanken kann für stationäre Rechner unter Verwendung von geeigneten Schnittstellen realisiert werden [GR92; Jep97]. Jedoch bildet die vergleichsweise geringe Rechenleistung von mobilen Geräten und deren unstetige Verbindung zum Netzwerk die Ursache dafür, dass diese Schnittstellen für mobile Umgebungen inkompatibel sind [SSPE04]. In Android wird auf Webdienste zurückgegriffen, die die Schnittstellen zu geteilten Datenbanken repräsentieren, wobei auch diese bei ausbleibender Netzwerkverbindung des mobilen Gerätes nicht erreichbar sind und die kollaborative Datenbearbeitung somit unmöglich machen.

Des Weiteren kann es bei kollaborativer Bearbeitung gemeinsam genutzter Daten zu Konflikten kommen, einhergehend mit Datenverlusten, die im jeweiligen Anwendungsbereich fatale Folgen mit sich bringen können.

1.2 Zielsetzung und Notwendigkeit

Die Lösung für die genannten Probleme erfordert eine verbesserte Architektur und Vorgehensweise für die Synchronisation der Daten, die sich in gemeinsam genutzten Datenbanken befinden. Hierbei darf die Datensicherheit nicht vernachlässigt werden.

Die vorliegende Arbeit setzt sich zum Ziel, bestehende Datensynchronisationsmechanismen zu analysieren und zu vergleichen, sowie ein neues Synchronisationsprotokoll einzuführen, welches auf mobilen Clients und Plattformen wie Android anwendbar ist und dem Benutzer gegenüber hohe Transparenz bietet. Dabei sollen die für einen solchen Prozess benötigten Datenmengen minimal gehalten werden und eine Bearbeitung gemeinsam genutzter Daten auch bei ausbleibender Netzwerkverbindung möglich sein. Außerdem muss der Benutzer weiterhin die Möglichkeit haben, bestimmte Daten von der Synchronisation zu entkoppeln, ausschließlich lokal zu halten und im Fall von Konflikten, diese selbst aufzulösen.

Um die Umsetzbarkeit der vorgestellten Konzepte nachzuweisen, stellt die vorliegende Arbeit eine ganzheitliche Implementierung eines Prototyps zur Verfügung.

1.3 Aufbau der Arbeit

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen beschreibt grundlegende Begriffe, die für die Verständnis der vorliegenden Arbeit benötigt werden.

Kapitel 3 – Anforderungsanalyse leitet anhand eines Anwendungsfalls aus dem eHealth-Umfeld die Anforderungen ab, die sich für ein Synchronisationsverfahren von mobilen Datenbanken in einem solchen Kontext ergeben.

Kapitel 4 – Verwandte Arbeiten analysiert und evaluiert bereits existierende Ansätze zur Synchronisation mobiler Datenbanken mit zentralisierten relationalen Datenbankmanagementsystemen.

Kapitel 5 – Lösungsansatz zeigt die verwendete Netzwerkarchitektur sowie eine Methodenbeschreibung für die gemeinsame Nutzung einer relationalen Datenbank unter Verwendung von mobilen Endgeräten.

Kapitel 6 – SSPMD-Protokoll stellt den systematischen Aufbau eines im Rahmen dieser Arbeit entwickelten Protokolls für die Synchronisation mobiler Datenbanken vor. Dabei werden client- und serverseitige Algorithmen entworfen und Datenstrukturen geschaffen, die einen effizienten Datenabgleich ermöglichen.

Kapitel 7 – Implementierung zeigt eine mögliche Implementierung des in Kapitel 6 vorgestellten Protokolls im Rahmen einer Anwendung aus dem eHealth-Umfeld.

Kapitel 8 – Evaluation evaluiert das Synchronisationsprotokoll und den entwickelten Prototyp sowohl in technischer als auch in anwendungsbezogener Hinsicht. Des Weiteren wird das vorgestellte Synchronisationsverfahren jenen aus den verwandten Arbeiten gegenübergestellt.

Kapitel 9 – Zusammenfassung und Ausblick fasst abschließend die Arbeit zusammen und stellt die Anknüpfungspunkte für zukünftige Arbeiten vor.

2 Grundlagen

In folgendem Kapitel werden grundlegende Konzepte der mobilen Datenhaltung und Probleme beschrieben, die für die Verständnis der vorliegenden Arbeit benötigt werden. In Abschnitt 2.1 wird zunächst das mobile Datenbanksystem eingeführt, gefolgt vom Begriff der gemeinsam genutzten Daten (Abschnitt 2.2). Anschließend behandeln Abschnitte 2.3 bis 2.5 Probleme, die bei gemeinsamer Nutzung einer Datenbank auftreten können. Dazu gehört mitunter die Dateninkonsistenz, das Lost-Update sowie die Entstehung von Datenkonflikten.

2.1 Mobile Daten und Datenbanksysteme

Android gehört zu den populärsten Betriebssystemen für mobile Endgeräte und bietet aus Entwicklersicht zwei Möglichkeiten für die persistente Datenspeicherung [PS11]. Dazu gehört zum Einen die direkte Nutzung des internen Dateisystems, um Dateien eines beliebigen Formats zu lesen oder zu schreiben. Zum Anderen können Daten mit Hilfe von *SQLite* in strukturierter Form abgelegt und mittels *Structured Query Language* (SQL) wieder abgerufen werden [Owe03]. Zweiteres besitzt den Vorteil, dass mitunter keine zusätzlichen Konzepte für die Serialisierung respektive Deserialisierung von Daten eingeführt werden müssen.

SQLite ist ein relationales Datenbankmanagementsystem, das bereits seit der ersten Version von Andorid integriert ist. Es charakterisiert sich insbesondere durch die einfache Handhabung und die ressourcenschonende Ausführung. Dessen Datenbanken werden in Form von `.db`-Dateien an einem beliebigen Ort abgespeichert und dort bearbeitet [RBB17]. Quelltext 2.1 zeigt ein Beispiel für das Öffnen und Bearbeiten einer solchen Datenbank unter Verwendung einer Kommandozeile.

Quelltext 2.1 Beispielhafte Konsolenausführung von *SQLite*

```
$ sqlite3 /home/user/product.db
SQLite version 3.16.2 2017-01-06 16:32:41
Enter ".help"for usage hints.
sqlite> create table product(name text, description text);
sqlite> insert into product values('soap dispenser', 'blue');
sqlite> select * from product;
soap dispenser|blue
sqlite>
```

Bei SQLite handelt es sich im Vergleich zu *Oracle*[1] oder *PostgreSQL*[2] nicht um einen Serverdienst, sodass keine entfernte Verbindungsanfragen durchgeführt werden können. Um eine gemeinsame Nutzung von strukturierte Daten zu ermöglichen, bedarf es daher eines neuen Konzepts, welches unter Verwendung von SQL-Anweisungen die Daten geräteübergreifend synchronisiert. Zwar könnten SQLite-Datenbanken mit üblichen Protokollen zur Datenübertragung auf der Dateiebene abgeglichen werden, allerdings ist in diesem Fall die Granularität zu niedrig, um zwei geänderte Datenbanken ohne einseitiger Datenverluste miteinander zu vereinen.

2.2 Gemeinsam genutzte Daten

Bei gemeinsam genutzten Daten handelt es im Kontext dieser Arbeit um Daten einer mobilen Anwendung, die geräteübergreifend bearbeitet werden können. Sie können auf zwei Weisen organisiert werden. Es besteht die Möglichkeit der direkten Verwendung einer globalen Datenbank, die über eine geeignete Schnittstelle angesprochen wird. Eine solche Schnittstelle wird zumeist mit Hilfe von Webservices, wie in Abbildung 2.1 dargestellt, realisiert [AHF+09]. Dieser Ansatz ist jedoch mit einem wesentlichen Nachteil insofern behaftet, dass weder der Abruf noch die Bearbeitung von gemeinsam genutzten Daten bei ausbleibender Netzwerkverbindung möglich sind.

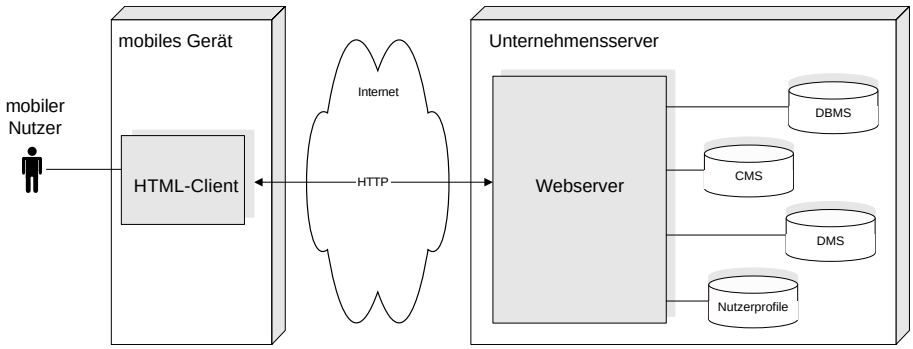


Abbildung 2.1: Client-Server-Architektur eines Unternehmensportals [ML02]

Andererseits besteht die Möglichkeit der Verwendung eines mobilen Datenbank-Abbilds, das auf Grundlage einer zentralisierten, gemeinsam genutzten Datenbank erstellt wird. Hierdurch wird eine dezentrale Modifikation von Daten unabhängig von der Verfügbarkeit der Netzwerkverbindung auf dem mobilen Endgerät ermöglicht. Dies ist jedoch zugleich die Ursache für die Entstehung der Dateninkonsistenz zwischen der clientseitigen und der gemeinsam genutzten Datenbank [CCP+10]. Diese Inkonsistenz kann mittels verschiedener Synchronisationsstrategien beseitigt werden, wobei mögliche Konflikte (Abschnitt 2.5) und Probleme wie Lost-Update (Abschnitt 2.4) behandelt werden müssen.

[1] https://www.oracle.com/de/database/
[2] https://www.postgresql.org/

2.3 Dateninkonsistenz

Ein Abbild einer relationalen Datenbank heißt inkonsistent, sobald es sich mit mindestens einem Datensatz von der zugrundeliegenden Datenbank unterscheidet. Eine solche Anomalie wird mit den Datenbank-Operationen `INSERT`, `UPDATE` oder `DELETE` bewirkt, die sich jeweils auf einen Datensatz beziehen und diesen neu hinzufügen, verändern oder löschen. Tabelle 2.1 stellt alle Fälle von Inkonsistenzen vor, die zwischen mehreren mobilen Clients und einer gemeinsam genutzten Datenbank auftreten können. Es ist hierbei davon auszugehen, dass die Daten ausschließlich auf den mobilen Clients modifiziert werden und es mehrere Fälle der Inkonsistenz innerhalb einer Datenbank geben kann. Fall 1 bis 3 beschreibt Modifikationen der Datenbank, die ausschließlich auf einem mobilen Client stattgefunden haben. Sie sind dabei auf der gemeinsam genutzten Datenbank und den Datenbanken anderweitiger Clients nicht angewendet, wodurch die Inkonsistenz zustandekommt. Fall 4 und 5 gibt konkurrierende Modifikationen der mobilen Datenbanken an. In Fall 4 ist ein und derselbe Datensatz auf mehreren Clients unterschiedlich geändert, während die gemeinsam genutzte Datenbank die unveränderte Version des betroffenen Datensatzes besitzt. Fall 5 beschreibt die Änderung eines Datensatzes, der auf einem konkurrierenden Client entfernt wird. Dabei ist auch dieser auf der gemeinsam genutzten Datenbank weiterhin vorhanden. Fall 6 bis 8 indiziert eine partielle Inkonsistenz. Dabei sind Modifikationen, die von einem mobilen Client stammen, auch auf der gemeinsam genutzten Datenbank angewendet. Allerdings steht dessen Anwendung auf den Datenbanken der übrigen Clients noch an.

Die Konsistenz ist wiederhergestellt, sobald alle durchgeführten Operationen eines Clients auch auf allen beteiligten Datenbanksystemen semantisch äquivalent angewendet werden.

Fall	Client_0	$\text{Client}_{i\in\{1,..,n\}}$	Gemeinsames RDBMS
1	`INSERT`	`NOC`	`NOC`
2	`UPDATE`	`NOC`	`NOC`
3	`DELETE`	`NOC`	`NOC`
4	`UPDATE`	`UPDATE`	`NOC`
5	`UPDATE`	`DELETE`	`NOC`
6	`INSERT`	`NOC`	`INSERT`
7	`UPDATE`	`NOC`	`UPDATE`
8	`DELETE`	`NOC`	`DELETE`

`NOC`: Keine Veränderung (no changes)

Tabelle 2.1: Inkonsistenz-Analyse in Bezug auf einen Datensatz (angelehnt an [CCP+10])

2.4 Lost-Update

Ein verlorenes Update (engl. *Lost-Update*) beschreibt im Allgemeinen den Verlust einer Änderung, bei dem eine Transaktion zwischen dem Lese- und Schreibvorgang einer konkurrierenden Transaktion das betroffene Datum modifiziert. Transaktionen sind in diesem Kontext als jene Operationen zu verstehen, die auf den mobilen Datenbanken durchgeführt werden. Die Lost-Updates treten zu dem Zeitpunkt auf, an dem die lokalen Datenbanken mit der gemeinsam genutzten Datenbank

wieder abgeglichen bzw. synchronisiert werden. Beispielhaft werden Datensätze aus einer gemeinsamen Datenbank auf zwei verschiedene Clients abgebildet, woraufhin ein und derselbe Datensatz beiderseits modifiziert wird. Nun werden die Änderungen vor dessen Bekanntgabe unter den beteiligten Clients jeweils mit der gemeinsam genutzten Datenbank nacheinander synchronisiert. Folglich werden zwei aufeinander folgende Operationen an einem Datensatz durchgeführt, wobei die Änderung der ersten Operation durch die der zweiten Operation überschrieben und somit ein Lost-Update verursacht wird.

2.5 Datenkonflikte

Wie die Fälle 4 und 5 aus Tabelle 2.1 bereits zeigen, werden Konflikte durch eine konkurrierende Modifikation eines gemeinsamen Datensatzes ausgelöst. Im Gegensatz zu Lost-Update treten diese nach einer Synchronisation bei den mobilen Datenbanken auf. Mit der Identifizierung von Konflikten, werden auch Lost-Updates implizit verhindert, da die Modifikation eines Datensatzes der gemeinsam genutzten Datenbank erst nach der Auflösung des Konflikts akzeptiert wird.

Im Allgemeinen wird zwischen den *Write/Write*- und den *Read/Write*-Datenkonflikten unterschieden. Bei *Write/Write*-Konflikten versuchen mindestens zwei Operationen einen und denselben Datensatz zeitgleich zu verändern. Bei *Read/Write*-Konflikten wird ein Datensatz von einem Client gelesen, während dieser von einem anderen Client zeitgleich verändert wird. Infolgedessen führt ein Client weitere Operationen auf Basis eines veralteten Datensatzes durch [TDP+94].

In der vorliegenden Arbeit werden beide Typen von Konflikten unter Zuhilfenahme einer geeigneten Versionierungsstrategie methodisch behandelt.

3 Anforderungsanalyse

In folgendem Kapitel wird ein Anwendungsfall anhand verschiedener Szenarien aus dem *eHealth*-Umfeld für das in dieser Arbeit vorgestellte Synchronisationsverfahren beschrieben (Abschnitt 3.1). Dabei wird mitunter der Bezug auf die im vorangegangenem Kapitel beschriebenen Probleme genommen. Anschließend werden hieraus Anforderungen extrahiert, die sich für die Synchronisation gemeinsam genutzter Daten in einem solchen Kontext ergeben (Abschnitt 3.2).

3.1 Anwendungsfall (eHealth)

Bei eHealth handelt es sich um die Verwendung der Informations- und Telekommunikationstechnologie, um die Gesundheit eines Menschen sowie die Abläufe im Gesundheitswesen zu verbessern [Eng02]. Dazu gehören u. a. Smart Devices, wie beispielsweise Smartphones oder Tablets, die in Verbindung mit einer Anwendung das gesundheitliche Personal oder gar einen Patienten bei bestimmten Aufgaben unterstützen.

Im konkreten Fall soll die Aufnahme und die Bearbeitung der in einer gesundheitlichen Einrichtung anfallenden Patientendaten (z. B. Behandlungsdaten) mit Hilfe einer mobilen Anwendung ermöglicht werden. Dazu besitzt das gesundheitliche Personal die entsprechende Anwendung auf einem mobilen Gerät, welches mitgeführt wird. Insbesondere besteht das Ziel darin, gewonnene Daten auf den mobilen Geräten bestimmter Personengruppen, die an einer Behandlung beteiligt sind, zu synchronisieren. Die Arbeit von F. Steimle et. al über *Enhancing Chronic patients' Health Online* (ECHO) beschreibt einen ähnlichen Anwendungsfall, bei dem die Kommunikation zwischen chronisch kranken Patienten und den behandelnden Ärzten verbessert werden soll.

3.1.1 Offline-Bearbeitung

Als erstes Szenario wird eine Notfallbehandlung gewählt, bei der ein Patient an einem spezifischen Ort Rettungshilfe anfordert. Diese wird per Straßenverkehr organisiert, wodurch die Konnektivität mobiler Geräte negativ beeinflusst wird. Die Ersthelfer können vor Ort nach der ersten Hilfe bzw. auf dem Rückweg zur Gesundheitsstelle den gesundheitlichen Zustand eines Patienten erfassen und ggf. eine grobe Diagnose anstellen. Die Gesundheitsstelle wird vorzeitig mit potentiell relevanten Informationen über den Patienten versorgt, wodurch bestimmte medizinischen Vorkehrungen und Entscheidungen sofort getroffen werden können.

3.1.2 Kollaboration

Die kollaborative Bearbeitung von Patientendaten bildet ein weiteres Szenario, bei dem mehrere gesundheitlichen Fachkräfte (FK) beteiligt sind (siehe Abbildung 3.1).

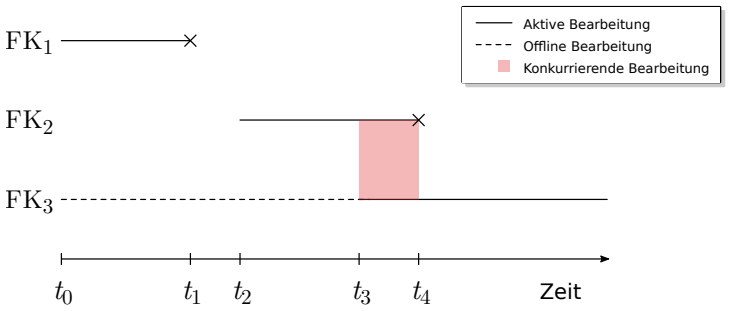


Abbildung 3.1: Ablauf einer kollaborativen Bearbeitung von Daten

Dabei werden zunächst große, gemeinsam genutzten Datenbestände auf das mobile Gerät von FK_1 geladen. Anschließend bearbeitet FK_1 einige dieser Daten und beendet zum Zeitpunkt t_1 die Synchronisation. Damit entsteht die in Abschnitt 2.3 vorgestellte Dateninkonsistenz. FK_2 startet zum Zeitpunkt t_2 die Synchronisation und bezieht alle durch FK_1 vorgenommenen Änderungen und setzt mit der Bearbeitung bis t_4 fort. FK_3 bearbeitet jedoch seit t_0 das lokale Abbild der gemeinsam genutzten Datenbank bei getrennter Netzwerkverbindung und beginnt erst bei t_3 mit der Synchronisation. Zwischen den Zeitpunkten t_3 und t_4 wird ein Datensatz von zwei Fachkräften zeitgleich bearbeitet, wodurch die Gefahr eines Lost-Updates (vgl. Abschnitt 2.4) zustandekommt. Mit solchen Situationen ist insbesondere bei der Übergabe von Patienten und beim Schichtwechsel des Personals in einer gesundheitlichen Einrichtung zu rechnen [HK12]. Außerdem bearbeitet FK_2 zum Zeitpunkt t_3 einen Datensatz, der in der Vergangenheit von FK_3 bereits gelöscht wurde, wodurch ein Datenkonflikt ausgelöst wird (vgl. Abschnitt 2.5).

3.1.3 Dateneinsicht

Die Patienten sind in der Lage mit Hilfe von lesenden Zugriffen auf die gemeinsam genutzte Datenbank eine Akteneinsicht selbst durchführen, um so den Überblick über die persönlichen Daten zu behalten. Auf diese Weise können sie auch Informationen, die beispielhaft die Medikamenteneinnahme betreffen, auch ohne die Anwesenheit eines Arztes vermittelt bekommen.

3.2 Anforderungen an das Synchronisationsverfahren

Aus den eingeführten Szenarien und aus dem Kontext der Anwendung können nun Anforderungen abgeleitet werden, die sich speziell an das Synchronisationsverfahren von Datenbanken auf mobilen Endgeräten richten.

A1 Zuverlässigkeit *(reliability)*:

Diese Eigenschaft beschreibt die korrekte Ausführung der Synchronisation; auch in einem Fehlerfall. Hierzu gehören Verbindungsabbrüche während der Datenübertragung, wie dem Szenario der Offline-Bearbeitung zu entnehmen ist. Insbesondere müssen die durch eine konkurrierende Bearbeitung (vgl. Abschnitt 3.1.2) auftretenden Konflikte erkannt und ohne Datenverluste behandelt werden.

A2 Effizienz *(efficiency)*:

Das Synchronisationsverfahren muss mit einer beschränken Datenübertragungsrate, die auf mobilen Endgeräten üblich ist, auskommen. Dabei muss die Menge an Metadaten, die für die Abwicklung der Synchronisation benötigt werden, möglichst minimal gehalten werden. Die Effizienz bezieht sich jedoch auch auf den Aspekt der Rechenintensität. Diese muss stets gering gehalten werden, da die mitgeführten mobilen Geräte eine eigene, stark beschränkte Energiequelle besitzen.

A3 Verfügbarkeit *(availability)*:

Gerade im Kontext des eHealths ist es von großer Bedeutung, dass die Bearbeitung von Daten nicht auf die Netzwerkanbindung angewiesen ist. Dies konnte insbesondere in Abschnitt 3.1.1 gezeigt werden. Der Benutzer muss auch in der Lage sein, lokale Daten unabhängig vom Stand der Synchronisation bearbeiten zu können.

A4 Transparenz *(transparency)*:

Die Synchronisation ist derart organisiert, dass eine Benutzerinteraktion hierfür weitestgehend ausgeschlossen ist. Damit werden die Benutzer der eHealth-Anwendung von zusätzlichen Aufgaben, die den Abgleich von Daten betreffen, entkoppelt.

A5 Datensicherheit *(security)*:

Da es sich in diesem Kontext überwiegend um persönliche Daten handelt, steht die Datensicherheit im Vordergrund. Dies betrifft insbesondere die Verwahrung von Daten auf den mobilen Endgeräten und in der gemeinsam genutzten Datenbank. Die lokal vorhandene Datensicherheit muss während der Synchronisation fortbestehen.

A6 Skalierbarkeit *(scalability)*:

Im beschriebenen Anwendungsfall kann es zu häufigen lesenden (vgl. Abschnitt 3.1.3) oder gar schreibenden (vgl. Abschnitt 3.1.2) Zugriffen auf die gemeinsam genutzten Datenbestände kommen. Aus diesem Grund muss das Synchronisationsverfahren auch bei einer großen Anzahl an mobilen Clients weiterhin korrekt ausgeführt werden.

4 Verwandte Arbeiten

Zahlreiche Arbeiten streben allgemeingültige Modelle zur Synchronisation von mobilen Daten an. Dabei wird des Öfteren auf klassische Client-Server-Architekturen zurückgegriffen, wobei es sich beim Server um eine zentrale Einheit handelt, die alle nötigen Komponenten für die Datenhaltung und die Synchronisation besitzt. Auch hier wird das Ziel verfolgt, die mobilen Knoten rechnerisch zu entlasten und die für eine Synchronisation benötigte Datenmenge weitestgehend zu reduzieren.

Um Änderungen an Datensätzen zu erfassen, werden verschiedene Strategien verfolgt. So basieren einige Arbeiten auf Hashing-Verfahren, die es erlauben, aufgrund des aktualisierten Hashwerts eines Datums, Änderungen am selben zu erkennen. Eine weitere, ähnliche Vorgehensweise benutzt Zeitstempel, um den temporalen Zusammenhang zwischen den Änderungen zu ermitteln. Auch gehören Schnappschüsse mitunter zu einer Strategie, um den Versionsstand eines Datensatzes zu referenzieren, wodurch sich Vorteile bei der Auflösung von Konflikten ergeben.

In diesem Kapitel werden verschiedene, bereits existierende Ansätze vorgestellt, analysiert und kurz evaluiert. Insbesondere wird auf bestehende Mechanismen zur Synchronisation mobiler Datenbanken mit zentralisierten relationalen Datenbankmanagementsystemen (RDBMS) eingegangen.

4.1 Hash-basierende Methoden

Bei einer *Hashfunktion* handelt es sich im Allgemeinen um eine nicht umkehrbare, surjektive Abbildung, die den Elementen einer Menge, Elemente aus einer wesentlich kleineren Menge zuordnet. Auf deren Grundlage baut auch die nachfolgend beschriebene Arbeit von M. Choi et al. [CCP+10] über *Synchronization Algorithms based on Message Digest* (SAMD) auf. Diese steht hier auch stellvertretend für alle Ansätze, die eine Erweiterung von SAMD bilden.

SAMD ist für die in Abbildung 4.1 gezeigte Netzwerkarchitektur konzipiert. Diese besteht einerseits aus mobilen Knoten (Clients), die keine durchgehende Netzwerkverbindung besitzen, und andererseits aus einem Synchronisationsserver mit einer direkten Verbindung zu einem Datenbankmanagementsystem (DBMS). Die Clients kommunizieren im Fall einer Synchronisation mit dem dafür vorgesehenem Synchronisationsserver. Dieser hat die Aufgabe, die im DBMS vorhandene Datenbank zu verwalten, darunter auch Tabellen, die für die Datenhaltung und effiziente Synchronisation nötig sind.

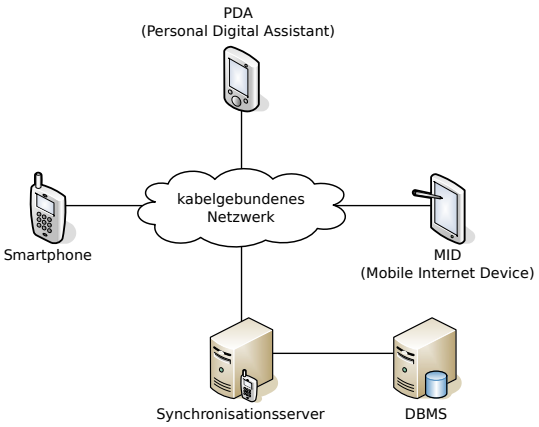


Abbildung 4.1: Netzwerkarchitektur des Anwendungsgebiets von SAMD [CCP+10]

4.1.1 Relationales Modell

In Abbildung 4.2 ist das Design der mobilen und der serverseitigen Datenbank, sowie die darin vorkommenden Beziehungen bezüglich der Synchronisation abgebildet. Um den Sachverhalt zu vereinfachen, werden die jeweiligen Kardinalitäten auf das Minimum reduziert und die relationale Struktur vereinfacht.

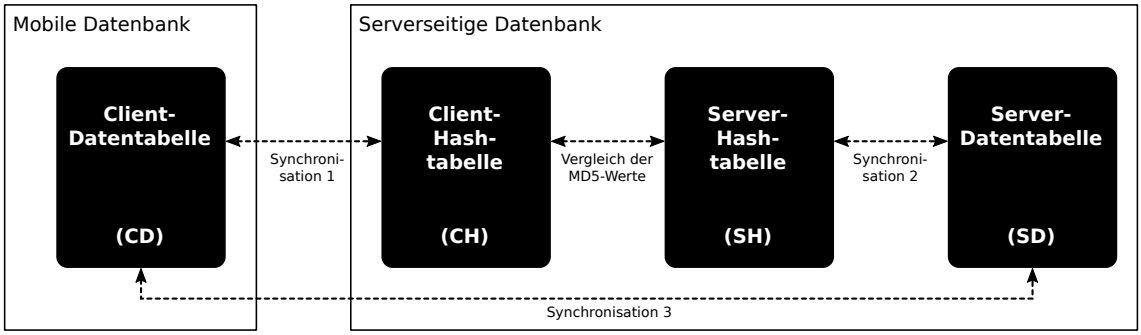


Abbildung 4.2: Datenbankdesign von SAMD (verändert nach [ATA13])

Die notwendigen Relationen werden anhand von Relationenschemata wie folgt erläutert.

CD ($\underline{id}, c_0, c_1, ..., c_n$) – Client-Datentabelle ist Bestandteil der Datenbank eines mobilen Knoten und enthält alle lokalen veränderlichen Daten eines Clients. Das Attribut *id* bildet den Primärschlüssel und $c_{i \in \{0,..,n\}}$ beinhaltet die durch den Benutzer veränderlichen Daten.

SD ($\underline{id}, c_0, c_1, ..., c_n$) – Server-Datentabelle befindet sich in der serverseitigen Datenbank und weist dieselbe Struktur wie *CD* auf. Sie bildet den zentralen Stützpunkt der Daten. Hierbei gilt für alle Datensätze: *SD.id = CD.id* .

CH ($\underline{id}, mdv, f$) – Client-Hashtabelle befindet sich in der serverseitigen Datenbank. Dessen Attribut *mdv* enthält für jedes Tupel die auf dem Client errechneten Hashwerte. Es gilt:

CH.id = CD.id . Das Attribut f beinhaltet boolesche Werte, die für die Markierung jener Tupel benutzt werden, dessen Synchronisation ansteht.

SH (*id*, *mdv*, f) – Server-Hashtabelle befindet sich ebenfalls in der serverseitigen Datenbank und ist analog zu *CH* definiert. Weiterhin gilt: *SH.id = CD.id* .

4.1.2 Bildung des Hashwerts

Hashwerte sind auf eine konstante Länge beschränkt und adressieren den Änderungszustand eines Tupels. Eine Änderung des Hashwerts schließt auf die Modifikation des zugehörigen Datums. Die Attributwerte $c_0, ..., c_n$ eines Tupels werden als Eingabe für die Hashfunktion benutzt.

So wird unter der Annahme, dass h eine geeignete Hashfunktion ist, für $n = 0$ der Hashwert $h(c_0)$ errechnet. Für $n > 0$ wird aus der Konkatenation $c_0 \circ c_1 \circ ... \circ c_n$ der Funktionswert von h berechnet.

4.1.3 Bedingungen

Sowohl client- als auch serverseitig sind die folgenden Bedingungen einzuhalten, damit die korrekte Ausführung von SAMD gewährleistet werden kann [CCP+10]:

1. Jede Relation muss einen Primärschlüssel besitzen.
2. Die Primärschlüssel von Daten- und Hashtabellen müssen für das jeweilige Tupel identisch sein.
3. Ein konkurrierendes Hinzufügen eines neuen Tupels sowohl in *CD* als auch in SD ist ausgeschlossen, da dies zur Kollision in der Vergabe des Primärschlüssels führen würde.

Aus der dritten Bedingung folgt, dass die Generierung der Primärschlüssel in den jeweiligen Datenbanksystemen nach einem globalen Paradigma erfolgen muss. Dies schließt u. a. die Wiederverwendung des Primärschlüssels eines gelöschten Tupels sowohl client- als auch serverseitig aus.

4.1.4 Algorithmus

SAMD gliedert sich in drei Prozeduren (Synchronisation 1, 2 und 3), die jeweils auf verschiedenen Tabellenpaaren angewendet werden. Hierbei können Synchronisation 1 und 2 parallel ausgeführt werden, während Synchronisation 3 von den Ergebnissen der beiden vorangegangenen Synchronisationsvorgängen abhängt.

Synchronisation 1

Die Daten der Tabellen *CD* und *SD* können unabhängig voneinander bearbeitet werden. Daher wird zunächst clientseitig eine temporäre Tabelle T bestehend aus der Spalte des Fremdschlüssels *id* und der Spalte h der jeweils errechneten $h(CD.c_0 \circ ... \circ CD.c_n)$ an den Server gesendet. Anhand der empfangenen Tabelle wird *CH* nach folgenden Instruktionen abgeglichen.

1. $\forall t \in T$ mit $t.id \notin CH.id$: Füge ein neues Tupel $(t.id,\ t.h,\ true)$ zur Tabelle *CH* hinzu.
2. $\forall u \in CH$ mit $u.id \notin T.id$: Setze $u = (u.id,\ NULL,\ true)$.
3. $\forall t \in T$ und $\forall u \in CH$ mit $t.id = u.id$ und $t.h \neq u.mdv$: Setze $u = (u.id,\ t.h,\ true)$.

t und u bezeichnen hierbei Tupel der Tabelle T und *CH*.

Synchronisation 2

Analog zu Synchronisation 1 werden hier die Hashwerte der Tabelle *SH* auf Grundlage von *SD* wie folgt errechnet.

1. $\forall v \in SD$ mit $v.id \notin SH.id$: Füge ein neues Tupel $(v.id,\ h(v.c_0 \circ ... \circ v.c_n),\ true)$ zur Tabelle *SH* hinzu.
2. $\forall w \in SH$ mit $w.id \notin SD.id$: Setze $w = (w.id,\ NULL,\ true)$.
3. $\forall v \in SD$ und $\forall w \in SH$ mit $v.id = w.id$ und $h(v.c_0 \circ ... \circ v.c_n) \neq w.mdv$: Setze $w = (w.id,\ h(v.c_0 \circ ... \circ v.c_n),\ true)$.

v und w bezeichnen hierbei Tupel der Tabelle *SD* und *SH*.

Synchronisation 3

Nun liegen die Tupel der Tabellen *CH* und *SH* so vor, dass sie miteinander verglichen werden können, um etwaige Änderungen und die Richtung der Synchronisation festzustellen. Dafür wird ein natürlicher Join auf *CH* und *SH* anhand der Primärschlüssel angewendet. Überschüssige Tupel werden zunächst ignoriert und nicht ins Join aufgenommen.

Alle Tupel, die den Wert *true* ausschließlich in der Spalte $CH.f$ besitzen, weisen auf eine clientseitige Modifikation hin, die auf die Datentabelle *SD* ebenfalls angewendet werden muss. Im Fall, dass *CH.mdv* den Wert *NULL* aufweist, wird das zugehörige Tupel aus allen Daten- und Hashtabellen entfernt. Andernfalls werden die Werte der Spalten $SD.c_i$ mit jenen aus der Tabelle *CD* überschrieben und der *mdv*-Wert der Tabelle *SH* für das entsprechende Tupel aktualisiert. Auf analoge Weise werden Änderungen an der Tabelle *CD* angewendet, im Fall dass das Tupel ausschließlich in der Spalte $SH.f$ den Wert *true* besitzt.

Alle Tupel, die den Wert *true* sowohl in der Spalte $CH.f$ als auch in $SH.f$ aufweisen, adressieren eine konkurrierende Änderung. Das heißt, dass die Richtung, in der die Synchronisation stattfindet, in diesem Fall nicht bestimmt werden kann. Daher stützt sich SAMD auf vordefinierte Policies, die entweder der client- oder der serverseitigen Datenbank Vorrang einräumen.

Alle Tupel, die in den Spalten *CH.f* und *SH.f* jeweils den Wert *false* besitzen, bedürfen aufgrund nicht vorhandener Änderungen keine Synchronisation.

Schließlich bleiben noch überschüssige Tupel, die sich beiderseits außerhalb des Joins befinden. Diese werden in die Tabelle *CD* bzw. *SD* hinzugefügt und somit ebenfalls synchronisiert. Nach jeder erfolgreichen Aktualisierung eines Tupels wird dessen boolescher Wert der Spalte f in allen Hashtabellen wieder auf *false* gesetzt.

Abbildung 4.3 zeigt eine mögliche Belegung von Hashtabellen, die für eine Abwicklung sämtlicher Schritte von SAMD ursächlich ist.

id	*mdv*	*f*
1	`00015a39`	*false*
2	`d7a8a214`	*true*
3	`445ded47`	*false*
4	*NULL*	*true*
5	`c6640708`	*true*
6	`b7496da2`	*true*

(a) *CH*

id	*mdv*	*f*
1	`00015a39`	*false*
2	`esa0ece0`	*false*
3	`e8a9cb79`	*true*
4	`d6f43399`	*false*
5	`bea96dae`	*true*

(b) *SH*

Abbildung 4.3: Mögliches Abbild von Hashtabellen zur Synchronisation mittels SAMD

4.1.5 Evaluation

Um das Volumen der zur Synchronisation benötigten Daten zu reduzieren, stützt sich SAMD auf den Austausch und den Vergleich von Hashwerten, woraus Informationen über den Änderungsstatus ermittelt werden können. Diese Methode entlastet insofern die mobile Datenbank, dass dort lediglich einfache und schnell ausführbare SQL-Operationen angewendet werden. Die Ausführung von Synchronisation 2 ist bei einer höheren Anzahl an mobilen Geräten ineffizient, da für jedes Tupel der Hashwert neu berechnet werden muss, wenn ein mobiler Client die Synchronisation anfordert [ATA13; CCP+10].

Die Anwendung dieser Methode ist für kontinuierliche Abfrage von Daten aus einer zentralen Datenbank nicht geeignet. Auch bei Aktualisierung eines Tupels müssten hierfür sämtliche Hashwerte neu berechnet und mit dem Server ausgetauscht werden. Des Weiteren fehlen jegliche Mechanismen zur Prävention von Datenverlusten. Beispielsweise können Hashwerte auf der serverseitigen Datenbank durch parallele Ausführung von Synchronisation 1 auf zwei mobilen Geräten unkontrolliert überschrieben werden. Daher eignet diese Synchronisationsmethode ausschließlich für nicht gemeinsam genutzte Datenbanken.

4.2 Timestamp-basierende Methoden

D. Sethia et al. stellen ein weiteres Verfahren vor, mit dem Tabellen mit multiplen Spalten zwischen mehreren mobilen Clients abgeglichen werden können. Auch hier werden die einzelnen

clientseitigen Abbilder der Datenbank auf Grundlage eines gemeinsam genutzten RDBMS synchronisiert. Hashwerte identifizieren lediglich ein Vorkommen einer Änderung, während anhand eines Zeitstempels sowohl der temporale Zusammenhang als auch die Reihenfolge einer Änderung festgestellt werden kann. Diesen Vorteil nutzen Sethia et al. aus und legen den *Mobile Replicated Database Management Synchronization*-Algorithmus (MRDMS) vor.

MRDMS wird hier im gesamten Abschnitt nach [SMC+14] beschrieben.

4.2.1 Datenmodell und -übertragung

Diese Methode verwendet ein Datenmodell, bei dem jede Daten-Tabelle, die zur Synchronisation vorgesehen ist, eine zugehörige Timestamp-Tabelle besitzt. Diese weist dabei gleiche Dimensionen wie die zugehörige Daten-Tabelle auf. Die Daten-Tabelle besteht aus einem Primärschlüssel und mehreren Spalten für die Benutzerdaten. Die Timestamp-Tabelle besteht aus einem Fremdschlüssel auf die Daten-Tabelle und den Spalten, die jeweils den letzten Änderungszeitpunkt in Form eines Zeitstempels aufnehmen. Damit besitzt jede Zelle der Daten-Tabelle eine zugeordnete Zelle aus der Timestamp-Tabelle. Sowohl die mobile als auch die gemeinsam genutzte Datenbank besitzt diese Tabellenpaare.

Die Daten werden in Form von dreidimensionalen Matrizen zwischen Client und Server ausgetauscht. Für dessen Realisierung kann beispielsweise das *JavaScript Object Notation*-Format (JSON) verwendet werden, da es eine beliebige Verschachtelung von Arrays erlaubt, wie in Quelltext 4.1 gezeigt wird.

Quelltext 4.1 JSON-Objektstruktur für den Austausch von Tabellen [SMC+14]

$$
\begin{array}{l}
[\,[\,[\text{Tabelle}_1\text{Zeile}_1],\ [\text{Tabelle}_1\text{Zeile}_2],\ \ldots,\ [\text{Tabelle}_1\text{Zeile}_n]\,],\\
\quad [\,[\text{Tabelle}_2\text{Zeile}_1],\ [\text{Tabelle}_2\text{Zeile}_2],\ \ldots,\ [\text{Tabelle}_2\text{Zeile}_n]\,],\\
\qquad\quad \vdots \qquad\qquad\quad \vdots \qquad\qquad\quad \vdots\\
\quad [\,[\text{Tabelle}_n\text{Zeile}_1],\ [\text{Tabelle}_n\text{Zeile}_2],\ \ldots,\ [\text{Tabelle}_n\text{Zeile}_n]\,]\,]
\end{array}
$$

Die Zeilen bestehen hierbei aus weiteren Arrays, die die jeweiligen Spalten repräsentieren. Die einzelnen Elemente des Tabellen-Arrays können auch den Wert *null* annehmen. Dieser kennzeichnet die Nichtexistenz von Änderungen an einer Tabelle.

4.2.2 Algorithmus

Mit dieser Vorgehensweise können unter Umständen mehrere Tabellen parallel synchronisiert werden. Um die Darstellung zu vereinfachen, wird hier der Algorithmus für die Synchronisation einer einzigen Daten-Tabelle zwischen dem Client und dem RDBMS beschrieben. *CD* bezeichnet hierbei die clientseitige Daten-Tabelle, während *SD* jene der gemeinsam genutzten Datenbank adressiert. Analog beschreiben *CTS* und *STS* die Timestamp-Tabelle der mobilen bzw. gemeinsam genutzten Datenbank.

Jeder Client besitzt eine lokale, persistente Variable *last_sync_ts*, die den Zeitpunkt der letzten Synchronisation mit der gemeinsam genutzten Datenbank festhält. Nach jeder erfolgreichen Synchronisation wird auch diese entsprechend aktualisiert.

Findet nun eine Veränderung einer Zelle in der Daten-Tabelle *CD* der mobilen Datenbank statt, wird der aktuelle Timestamp an der zugehörigen Position in der Timestamp-Tabelle gesetzt.

Zu Beginn der Synchronisation wird zunächst die clientseitige Timestamp-Tabelle *CTS* in das JSON-Format überführt (siehe Abbildung 4.4). Anschließend wird zusätzlich der Wert des *last_sync_ts* in das JSON-Objekt aufgenommen und an den Server der gemeinsam genutzten Datenbank gesendet.

id	t_0	t_1
1	1521000000	0
2	0	1521001000

(a) *CTS*

```
{
  C: [
    {"1", "1521000000", "0"},
    {"2", "0", "1521001000"}
  ],
  last_sync_ts: "1521000001"
}
```

(b) JSON-Objekt von *CTS*

Abbildung 4.4: Beispiel für eine Timestamp-Tabelle und dessen JSON-Objekt

Zellen mit dem Timestamp-Wert 0 weisen unveränderte Zellen in *CD* auf. Die Änderung von mindestens einer Zelle führt zur Aufnahme der gesamten Zeile in die Matrix C. Für den Fall, dass es keine Änderungen gibt, wird lediglich die Variable *last_sync_ts* an den Server übermittelt.

Nun wird auf dem Server das empfangene JSON-Objekt (Matrix C und *last_sync_ts*) mit der dort vorhandenen Tabelle *STS* abgeglichen. Dabei wird eine temporäre, gleichdimensionale Matrix T erzeugt, die wie folgt belegt wird.

1. Alle Werte aus der Tabelle *SD*, dessen zugehörige Zellen aus *STS* einen größeren Zeitstempel als *last_sync_ts* besitzen, werden in T geschrieben.
2. Andernfalls wird das Zeichen „#“ in die zugehörige Zelle von T gesetzt, wodurch eine nötige Aktualisierung in *SD* signalisiert wird.
3. Für jene Zellen aus C, die den Wert 0 besitzen, wird in T das Zeichen „@“ gesetzt, falls es für diese keine Änderung in *SD* gegeben hat. Ansonsten wird auch hier der konkrete Wert der jeweiligen Zelle aus *SD* übernommen.

Die so belegte Matrix T – auch Requirement-Matrix genannt – wird nun zusammen mit dem aktuellen Zeitstempel *current_server_ts* zurück zum mobilen Client gesendet. Abbildung 4.5 zeigt ein Beispiel in Form eines JSON-Objekts.

Nach dem Empfang von T auf dem mobilen Client, werden alle Werte, die weder „@“ noch „#“ gleichen, in die Tabelle *CD* aufgenommen. Die Matrix T wird nun zu T' modifiziert, indem die mit „#“ markierten Zellen mit den aktuelleren Werten aus *CD* belegt werden. Anschließend wird *last_sync_ts* dem empfangenem Timestamp *current_server_ts* angeglichen. Schließlich wird T' erneut an den Server gesendet, wobei dort die angeforderten Werte in *DS* aktualisiert werden.

```
{
  T: [
    {"1", "bbab", "@"},
    {"2", "babb", "#"}
  ],
  current_server_ts: "1521002000"
}
```

Abbildung 4.5: Beispiel einer Requirement-Matrix als JSON-Objekt

Um die korrekte Vergabe der Primärschlüssel beim Hinzufügen eines neuen Datensatzes zu gewährleisten, müssen diese durch das gemeinsam genutzte RDBMS generiert und an den mobilen Client gesendet werden. Dabei wird ein neuer Datensatz sowohl in *SD* als auch in *STS* erzeugt, wobei die Zellen zunächst mit initialen Werten belegt werden. Nach der Aktualisierung des Primärschlüssels in *CD* können die Änderungen weiter nach der bisherigen Vorschrift synchronisiert werden.

Abbildung 4.6 visualisiert zusammenfassend den Synchronisationsvorgang und den Datenfluss zwischen Client und Server während der Ausführung von MRDMS.

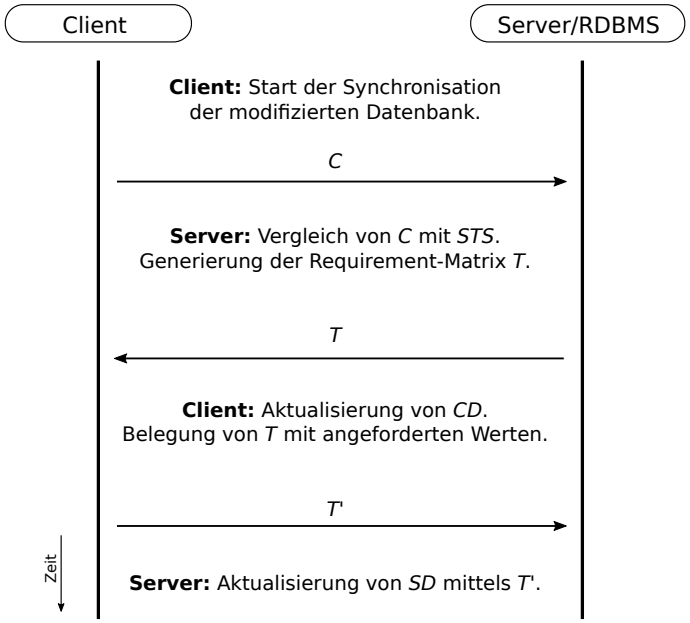


Abbildung 4.6: Datenfluss in MRDMS [SMC+14]

4.2.3 Evaluation

MRDMS weist gegenüber SAMD eine enorme Verbesserung in der Effizienz bezüglich des zu übertragenden Datenvolumens auf, da zunächst nur Informationen, die eine Zellenänderung identifizieren, gesendet werden. Dies bezieht sich insbesondere auf die Matrizen, die die Zeitstempel der jeweiligen Zellen aus der Daten-Tabelle beinhalten. Des Weiteren ist unter Verwendung

von MRDMS die Rechenintensität auf dem mobilen Client sehr niedrig, wodurch der Stromverbrauch reduziert wird. Dies ist ebenfalls auf die ressourcenschonende Verwendung der mobilen Datenbank zurückzuführen.

Allerdings ist bei diesem Verfahren eine Auflösung von auftretenden Konflikten zweier Änderungen an derselben Zelle sehr schwierig zu realisieren [SMC+14]. Sethia et al. deuten auf die Kompatibilität des Verfahrens für die serverseitige Behandlung von Konflikten. Jedoch impliziert dies lediglich rudimentäre Strategien zur Konfliktauflösung, wie beispielsweise die Priorisierung einer bestimmten, mobilen Datenbank.

Mit Hilfe von Timestamps können Datenverluste, für die der Benutzer selbst nicht verantwortlich ist, weiterhin nicht vermieden werden. Die Ursache hierfür liegt in der Bedingung, dass eine Datenzelle mit einem niedrigeren Zeitstempel durch jene mit einem höheren Zeitstempel überschrieben werden kann. Folglich kann ein Wert, dessen Zeitstempel sich zwischen den beiden genannten Extrema befindet, unkontrolliert verworfen werden.

4.3 Andere Verfahren

Die zuvor vorgestellten Verfahren gleichen eine lokale mit einer gemeinsam genutzten Datenbank anhand von korrespondierenden Algorithmen ab. Damit werden bereits mobil durchgeführte Modifikationen an Datensätzen unter Erfüllung von Vorgaben auch auf der gemeinsam genutzten Datenbank direkt angewendet. Allerdings müssen sich Synchronisationsmechanismen nicht auf die direkte Übermittlung von Datensätzen zwischen zwei Datenbanken beschränken. So können die auf mobilen Clients durchgeführten Änderungen als Transaktionen aufgefasst werden. Diese schließen erst bei einer aktiven Verbindung zum gemeinsamen RDBMS ab. Sie beinhalten Schritte, die als eine Einheit oder gar nicht ausgeführt werden. Auf diese Weise kann das gemeinsame RDBMS Zusicherungen an den mobilen Client erteilen, um Konflikte oder Inkonsistenzen zu vermeiden. Hierdurch kann z. B. ein neuer Datensatz in der gemeinsam genutzten Datenbank hinzugefügt werden und dort verbleiben, nur wenn der mobile Client, auf dem die Änderung durchgeführt wurde, den Primärschlüssel erhält. In diesem Fall sichert das RDBMS einen Primärschlüssel zu. Falls eine Transaktion, beispielsweise bedingt durch den Verlust der Netzwerkverbindung, nicht durchgeführt werden kann, können zwei Maßnahmen zur Erhaltung der Konsistenz angewendet werden.

1. Die bis zum Kommunikationsabbruch vorgemerkten Transaktionsschritte werden nicht ausgeführt und anschließend verworfen.
2. Alle bis zum Kommunikationsabbruch durchgeführten Transaktionsschritte werden rückgängig gemacht.

Bei Modifikationen während der ausbleibenden Netzwerkverbindung können auch vorläufige Transaktionen ausschließlich auf dem mobilen Client durchgeführt werden. Somit wird die Kommunikation mit der serverseitigen Synchronisationseinheit (einschließlich RDBMS) simuliert, wobei provisorische Daten anfallen können [Höp01]. Bei einer neuen Netzwerkverbindung werden die vorläufigen Transaktionen wiederaufgenommen und abgeschlossen. Nachteilhaft hierbei ist die Implementierung des Verhaltens der serverseitigen Synchronisationseinheit auf dem mobilen

Client, wodurch die Ressourcen und die Rechenintensität des mobilen Geräts stärker beansprucht werden.

Als Erweiterung hierzu können s. g. Schnappschüsse (engl. *Snapshots*) zur Auflösung von bestimmten Konflikten verwendet werden. Phatak et al. [PN04] stellen hierzu eine Methode vor, die sich der Aufbewahrung mehrerer Versionen bzw. Snapshots eines Datensatzes bedient. Ein Snapshot enthält mehrere Attributwerte eines Datensatzes, die unabhängig voneinander versioniert sind, und wird erst nach dem Abschluss einer Transaktion erstellt. Beim Änderungskonflikt eines Datensatzes werden diese Snapshots für den Versuch einer autonomen Auflösung des Konflikts benutzt, indem konfliktfreie Attribute miteinander zu einem neuen Datensatz verschmolzen werden. Im Beispiel aus [Höp01; PN04] wird ein solcher Akt wie folgt veranschaulicht.

Gegeben sei ein Datensatz D mit den Attributwerten x und y, sowie zwei konkurrierende Clients C_1 und C_2. Nun wird D anhand einer Transaktion von C_1 mit der Belegung $\{x = 1, y = 1\}$ auf dem gemeinsam genutzten RDBMS geschrieben. Unmittelbar danach bildet C_2 den Datensatz D in seine lokale Datenbank ab und trennt die Verbindung zum gemeinsamen RDBMS. Daraufhin schreibt C_1 die Attributwerte $\{x = 2, y = 1\}$ und schließt die Transaktion ab, während C_2 eine lokale Transaktion mit $\{x = 1, y = 3\}$ tätigt. Bei der nächsten Synchronisation durch C_2 kommt es zu einem Änderungskonflikt auf D, da der Attributwert x durch C_1 bereits modifiziert wurde. Abbildung 4.7 visualisiert die beschriebene Situation.

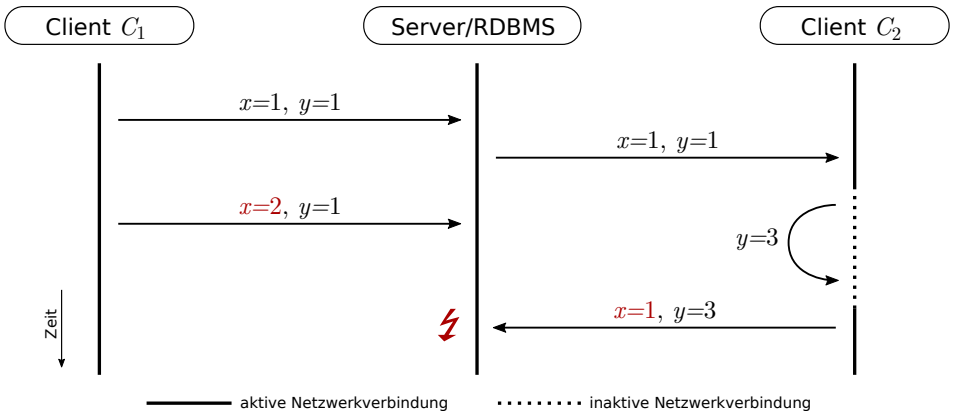


Abbildung 4.7: Datenfluss bei einem Änderungskonflikt

Nun besitzt der Server bis zum Auftritt des Konflikts zwei Snapshots von D mit versionierten Attributwerten. Diese sind $(x_0 = 1, y_0 = 1)$ und $(x_1 = 2, y_0 = 1)$. Anhand dessen können veraltete Attributwerte aus der letzten Transaktion von C_2 identifiziert werden. Folglich wird ein neuer Datensatz mit $\{x = 2, y = 3\}$ aus x_1 und y_1 generiert. Dieser löst den Konflikt zwischen $x = 2$ und $x = 1$ und wird in den gemeinsam genutzten Datenbank zur Aktualisierung benutzt. Ebenfalls wird erneut ein Snapshot $(x_1 = 2, y_1 = 3)$ erstellt, womit nachfolgende Konflikte gelöst werden können.

Dieses Verfahren ist erfolgreich, solange nicht dieselben Attributwerte von mehreren Clients mit inaktiver Netzwerkverbindung modifiziert werden. Daher eignet sich dieser Ansatz bei sehr feingranularen Datensätzen, die aus vielen Attributwerten bestehen. Die Wahrscheinlichkeit des Auftretens eines Änderungskonflikts an einem Attributwert wird somit niedriger, je mehr Attribute vorhanden sind.

5 Lösungsansatz

Im vorangegangenem Kapitel wurden bereits existierende Ansätze für die Synchronisation von mobilen Datenbanken vorgestellt. Nachfolgend wird die Umgebung beschrieben, in der das in dieser Arbeit eingeführte Synchronisationsverfahren zum Einsatz kommt (Abschnitt 5.1). Dabei wird mitunter auf die Datensicherheit eingegangen und die allgemeine Methode des Synchronisationsverfahrens vorgestellt (Abschnitt 5.2).

5.1 Netzwerkarchitektur

Diese Vorgehensweise ist weiterhin für ein Client-Server-Modell ausgelegt, bei dem der Client die Rolle eines mobilen Gerätes und der Server die Rolle eines Synchronisationsservers übernimmt. Dabei ist der Synchronisationsserver selbst unabhängig vom Ort des relationalen Datenbankmanagementsystems (RDBMS) und umgekehrt. Daher ist es möglich, diesen an einem beliebigen Ort in der Cloud zu platzieren, wie in Abbildung 5.1 verdeutlicht wird.

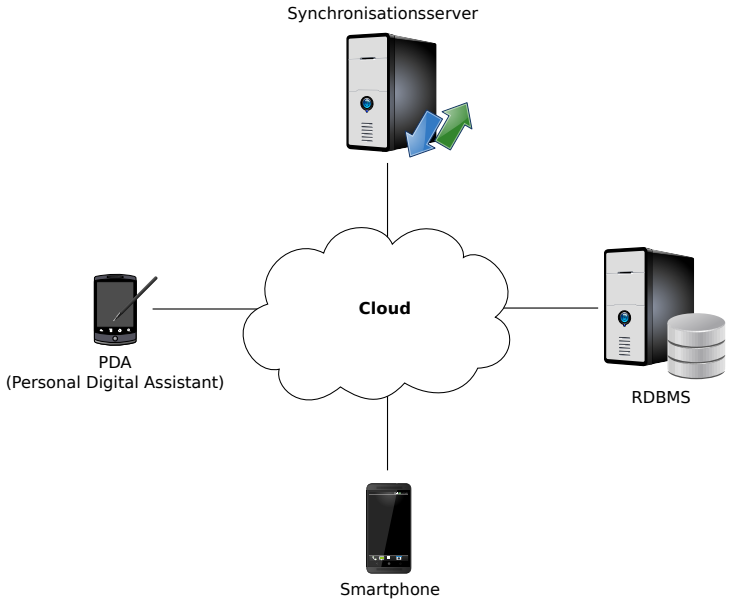


Abbildung 5.1: Beispiel einer Cloud-Architektur

Dadurch dass der Client die Datenbankverbindung definiert, ist es auch möglich, dass mehrere Clients jeweils Daten aus unterschiedlichen Datenbanksystemen über denselben Synchronisationsserver simultan synchronisieren.

5.1.1 Synchronisationsserver

Bei einem Synchronisationsserver handelt es sich um einen Dienst, der vorzugsweise auf stationären Rechnern ausgeführt wird. Dieser dient als Brücke zwischen den mobilen Clients und dem RDBMS. Der Server ist in der Lage, mit Hilfe von geeigneten Schnittstellen eine Verbindung zu einem externen RDBMS herzustellen und Datenbankabfragen durchzuführen. Dazu gehört das Anlegen neuer Datenbanken sowie das Hinzufügen, Löschen oder Verändern von Datensätzen. Hier werden Client-Anfragen zur Synchronisation von Datensätzen bearbeitet.

5.1.2 Mobiler Client

Mobile Clients stellen Anwendungen (Apps) auf einem portablen Gerät dar. Sie besitzen die Anbindung an eine lokale Datenbank, die vom Benutzer implizit bearbeitet werden kann. Essentiell für eine transparente Synchronisation ist ein lokaler und ressourcenschonender Hintergrunddienst, von dem Änderungen an Daten erfasst werden können, die dann schließlich an den Synchronisationsserver übermittelt werden. Zeitgleich kann eine Änderung, die ihren Ursprung bei einem konkurrierenden mobilen Client hat, empfangen und lokal angewendet werden.

5.1.3 Datensicherheit

Da bei diesem Synchronisationsmechanismus die Daten mehrere Instanzen beginnend mit der mobilen Datenbank über das Netzwerk passieren können, darf zu keinem Zeitpunkt die Datensicherheit vernachlässigt werden. Für die sichere lokale Verwahrung der Daten auf mobilen Endgeräten wurde die *Privacy Management Platform* (PMP) [SM13; SM14] in Kombination mit dem *Secure Data Container* (SDC) [SM15; SM16] oder dem *CURATOR* [SM18] eingeführt. Unter Benutzung dieser Konzepte ist es möglich, Daten in einer lokalen Datenbank zu verschlüsseln und entsprechende Berechtigungen an Anwendungen zu vergeben, die darauf zugreifen dürfen. In diesem Kontext ist zum Einen die App des Benutzers, mit der die Daten verändert werden, und zum Anderen der Dienst, der für die Synchronisation zuständig ist, betroffen. Ebenso muss der Datenverkehr zwischen dem mobilen Client und dem Synchronisationsserver verschlüsselt werden. Hierzu können übliche, bisher sichere Verfahren wie *Transport Layer Security* (TLS) [Tur14] verwendet werden.

5.2 Synchronisationsmethode

Dieser Lösungsansatz verfolgt das Ziel, ein System für die Datenhaltung auf multiplen mobilen Geräten zu ermöglichen, wobei potentielle Dateninkonsistenzen ausgeräumt werden. Hierfür besitzt jeder Client eine eigene relationale Datenbank, die stets mit der zentralisierten Datenbank synchronisiert wird. Eine Datenänderung muss daher zwangsläufig den Synchronisationsserver passieren, bevor sie auf den restlichen Clients, die an der Synchronisation beteiligt sind, angewendet werden kann.

Für die client- und serverseitigen Daten bietet sich ein relationales Schema an, mit dessen Hilfe zwischen mehreren Fällen und der Richtung der Synchronisation unterschieden werden kann. Die Modifikation, die Löschung oder das Hinzufügen eines Datensatzes gehört zur Änderung der Datenbank, die auf einem mobilen Client entweder bezogen oder gesendet wird. Dabei müssen gewisse Regeln während des Synchronisationsvorgangs eingehalten werden, um Datenverluste (vgl. Abschnitt 2.4) zu verhindern.

Die Lösung beinhaltet mitunter die sofortige Eliminierung der Dateninkonsistenz nach Eintritt der Änderung an der mobilen Datenbank. Dieser Vorgang hängt jedoch von der Verfügbarkeit der Netzwerkverbindung auf dem mobilen Gerät ab und kann unter Umständen nicht durchgeführt werden. Des Weiteren existiert ein weiterer Synchronisationsmechanismus, der alle Änderungen, die bei fehlender Konnektivität des mobilen Gerätes entstanden sind, anhand einer Versionierungsstrategie effizient synchronisiert. Dies Umfasst die Änderungen sowohl an der mobilen als auch der zentralisierten Datenbank. Es werden ebenfalls Änderungen an Datensätzen erkannt, die mit den Änderungen eines weiteren Benutzers in Konflikt stehen (vgl. Abschnitt 2.5). Für dessen Auflösung wird eine weitere Versionierungsstrategie angewendet, die den normalen Verlauf der Synchronisation wieder ermöglicht.

Die Realisierung dieser Methode erfordert ein vordefiniertes Protokoll, welches sowohl client- als auch serverseitig befolgt wird, worunter sowohl die zu synchronisierenden Daten als auch Informationen bezüglich der Synchronisation in beide Richtungen ausgetauscht werden.

6 SSPMD-Protokoll

Im Allgemeinen legen Protokolle fest, welche voneinander abgängige Schritte von zwei Kommunikationspartnern durchzuführen sind, um ein bestimmtes Ziel zu erreichen. Sie finden Vertretung im Informationsaustausch wie beispielsweise bei der Dateiübertragung oder Authentifizierung. Auch kann ein Protokoll den Ablauf einer Synchronisation von Daten zwischen zwei Instanzen, denen Datenbanken zugrundeliegen, vorschreiben [AST02]. Im Folgenden wird der Aufbau eines zustandsbasierenden Synchronisationsprotokolls für mobile Datenbanken (engl. *State based Synchronization Protocol for Mobile Databases*, SSPMD) unter Beachtung der in Kapitel 3 eingeführten Anforderungen vorgestellt.

Abschnitt 6.1 führt zunächst das relationale Modell der zugrundeliegenden Datenbanken ein. Abschnitt 6.2 stellt Regeln auf, nach denen die Daten in der entsprechenden Relation verändert werden dürfen. In Abschnitt 6.3 wird das Nutzungsprinzip von Ereignissen und Aufgaben erläutert, während die Abschnitte 6.4 bis 6.6 das theoretische Konzept der Synchronisation sowie die zugrundeliegenden Algorithmen beschreiben.

6.1 Relationsschema

Der vorgestellte Ansatz ist für Daten ausgelegt, die sowohl server- als auch clientseitig in Form von *Key-Value*-Paaren gespeichert werden, wobei diese mit zusätzlichen Metadaten versehen sind. Abbildung 6.1 zeigt das Relationsschema der gemeinsam genutzten Datenbank.

SHARED_DATA_RECORDS			
SID	VERSION	TIMESTAMP	CONTENT
INT			TEXT

Abbildung 6.1: Relationsschema der gemeinsam genutzten Datenbank im RDBMS

Der Primärschlüssel `SID` und das Attribut `CONTENT` stehen hierbei stellvertretend für die *Key-Value*-Paare. Das Attribut `VERSION` beinhaltet die Versionsnummer eines Datensatzes, die ausschließlich auf den mobilen Clients erhöht wird. `TIMESTAMP` hält den genauen Zeitpunkt der Erstellung oder Veränderung eines Datensatzes fest, während `CONTENT` das konkrete Datum in Form einer Zeichenkette beinhaltet. Diese ermöglicht die Speicherung von beliebig komplexen Objekten, die beispielsweise mittels JSON, YAML oder XML serialisiert sind.

Eine ähnliche Struktur weist auch die Relation des mobilen Clients auf (siehe Abbildung 6.2). Dabei bildet `CID` den Primärschlüssel, während hier das Attribut `SID` implizit als Fremdschlüssel der Tabelle `SHARED_DATA_RECORDS` dient, da sich diese nicht auf demselben Datenbanksystem befindet.

CLIENT_DATA_RECORDS					
CID	SID	MODE	VERSION	TIMESTAMP	CONTENT
INT					TEXT

Abbildung 6.2: Relationsschema für die Daten in der mobilen Datenbank

Des Weiteren beinhaltet `MODE` den kodierten Modus, in dem sich ein Datensatz befinden kann. Dieser kann einen der fünf Werte annehmen kann, die wie folgt beschrieben werden.

NEW – Ein Datensatz, der vom Benutzer neu hinzugefügt wurde und für die Synchronisation bereit ist.

RELEASED – Ein synchronisierter Datensatz, der für eine weitere Änderung durch den Benutzer oder den Synchronisationsserver freigegeben ist. Dieser kann durch den Synchronisationsserver auch endgültig gelöscht werden.

MODIFIED – Ein Datensatz, der durch den Benutzer lokal verändert wurde. Dieser wird erst nach einer erfolgreichen Synchronisation wieder in den Modus *RELEASED* überführt. Andernfalls verbleibt der Modus auch nach erneuter Veränderung des Datums.

DELETED – Ein Datensatz, dessen Löschvorgang vom Benutzer eingeleitet wurde. Dieser wird erst dann vollständig gelöscht, sobald die Löschung auch durch den Synchronisationsserver veranlasst wurde.

OFFLINE – Dieser Datensatz wurde seitens des Benutzers von der Synchronisation entkoppelt und ist nur zur lokalen Bearbeitung verfügbar. Dabei löscht der Synchronisationsserver dieses Datum aus der gemeinsam genutzten Datenbank und aus Datenbanken konkurrierender Clients. Die Möglichkeit der Überführung in den Modus *NEW* bleibt jedoch bestehen.

Dadurch dass bei dieser Vorgehensweise die Möglichkeit besteht, Daten auf mobilen Clients bei getrennter Verbindung zum Synchronisationsserver zu bearbeiten, kann es unter Umständen zu Konflikten kommen. Auch hierfür steht eine geeignete Relation für die Datenbank des mobilen Clients bereit, in der zwei Typen von Konflikten verzeichnet werden können (siehe Abbildung 6.3).

CLIENT_CONFLICT_RECORD					
CID	SID	CONFLICT_TYPE	VERSION	TIMESTAMP	CONTENT
INT					TEXT

Abbildung 6.3: Relationsschema für die Konflikte in der mobilen Datenbank

Die Definitionen für das Relationsschema aus Abbildung 6.2 entsprechen mit Ausnahme des Attributs `CONFLICT_TYPE` auch hier. Dabei kann `CONFLICT_TYPE` eines der nachfolgend beschriebenen Werte annehmen.

MODIFICATION – Ein Konflikt, bei dem ein veralteter Datensatz durch den Benutzer modifiziert wird.

PRESENCE – Ein Konflikt, bei dem ein Datensatz durch den Benutzer modifiziert wird, während dieser in der gemeinsam genutzten Datenbank nicht mehr vorhanden ist.

Im Fall eines Konfliktes wird der betroffene Datensatz zu dieser Relation kopiert bzw. verschoben, wie in den nachfolgenden Abschnitten dieser Arbeit genauer beschrieben wird.

6.2 Versionierung

Die Zahl der Version dient zur Markierung des Versionsstandes eines Datensatzes. Sie bildet einerseits die Grundlage für eine effiziente Synchronisation und verhindert dabei mögliche Datenverluste.

Beim Hinzufügen eines neuen Datensatzes `A` in die Relation `CLIENT_DATA_RECORDS` der mobilen Datenbank wird eine initiale Versionsnummer vergeben. Diese wird bei jedem Wechsel des Modus von *RELEASED* zu *MODIFIED* inkrementiert (siehe Bedingung 6.1).

$$\texttt{A.VERSION} = \texttt{A.VERSION} + 1 \tag{6.1}$$

Für eine Veränderung respektive Überschreibung eines in der Relation `SHARED_DATA_RECORDS` vorhandenen Datensatzes `A` durch `A'` muss die folgende Bedingung erfüllt werden.

$$\texttt{A.VERSION} = \texttt{A'.VERSION} - 1 \tag{6.2}$$

Diese Bedingung beugt das Problem eines Lost-Updates bei konkurrierender Modifikation eines Datensatzes vor. Die Nichterfüllung dieser Bedingung führt zu einem Änderungskonflikt, der in die dafür vorgesehene Relation `CLIENT_CONFLICT_RECORD` verzeichnet wird. Dies ist insbesondere der Fall, wenn ein Datensatz eines mobilen Clients getrennt vom Synchronisationsserver geändert wird, wobei derselbe Datensatz von einem anderen Client geändert und mit der gemeinsam genutzten Datenbank synchronisiert wurde.

6.3 Ereignisse und Aufgaben

Ereignisse (engl. *Events*) entstehen aus Aktionen, die vom Benutzer durchgeführt werden, um ein neues Datum zu erzeugen, zu verändern oder zu entfernen. Ein Ereignis kann dabei nur einen Datensatz betreffen. Sie werden zur Verfolgung von Änderungen an der mobilen Datenbank und damit zur sofortigen Synchronisation verwendet, wie in Abbildung 6.4 gezeigt wird.

Aus Ereignissen werden Aufgaben generiert. Diese beinhalten den betroffenen Datensatz, sowie eine Aktion in Form eines Wertes, die darauf angewendet werden soll. Die Aufgaben werden dann in eine Warteschlange eingereiht (1), mit dessen Abarbeitung die entsprechende Änderung zunächst an den Synchronisationsserver übermittelt wird (2). Daraufhin wird sie in der gemeinsam

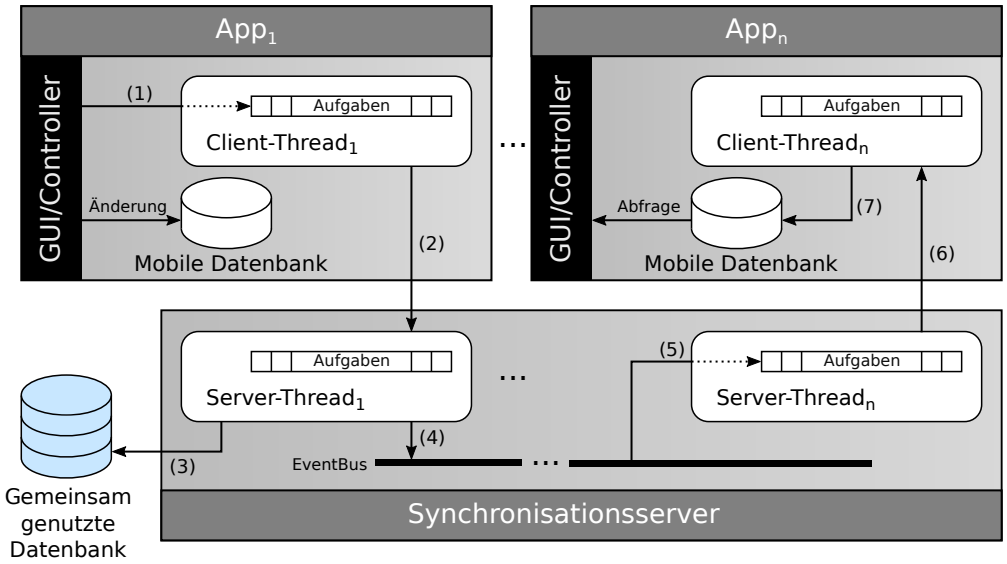


Abbildung 6.4: Nutzungsprinzip von Ereignissen und Aufgaben

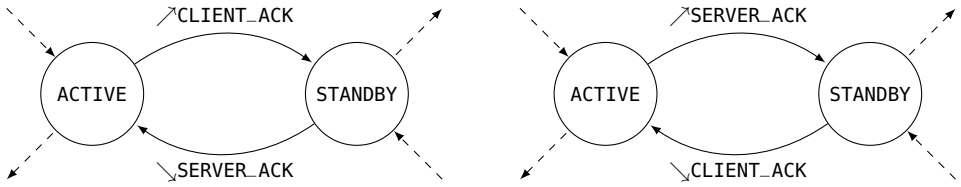
genutzten Datenbank angewendet (3). Nach der Modifikation der Datenbank erzeugt der Synchronisationsserver ein lokales Benachrichtigungsereignis, das mit Hilfe des Observer-Patterns (z. B. *EventBus*) an alle anderen Server-Threads übermittelt wird (4). Das Benachrichtigungsereignis wird anschließend in eine Aufgabe umgewandelt und in die Warteschlange der jeweiligen Server-Threads hinzugefügt (5). Schließlich wird die in der Aufgabe beschriebene Änderung auch an die zugehörigen Clients gesendet (6) und dort angewendet (7). Für eine schleifenfreie Übermittlung von Ereignissen sind Identifikatoren unerlässlich. Diese werden dazu benutzt, um jenen Thread bzw. Sender, aus dem ein Ereignis hervorgeht, zu identifizieren und somit eine Rückkopplung zu unterbinden.

6.4 Synchronisationsautomat

Ein sequenzielles Protokoll vergleichbar mit HTTP [FGM+99] oder FTP [PR85] kann für einen stetigen Datenaustausch nicht verwendet werden, da wiederkehrende Neuausführungen zur Ineffizienz führen. Dies betrifft insbesondere das Zeitverhalten und die zu übertragende Datenmange. Daher bietet sich in diesem Kontext die Verwendung von *deterministisch endlichen Automaten* (DEA) an. Hierbei hängen die zu Durchführung des Protokolls nötigen Sequenzen von Zuständen und nicht länger zwingend von ihren vorhergehenden Sequenzen ab. Dabei besitzt sowohl der mobile Client als auch der Synchronisationsserver eine interne Simulation eines DEA, mit dessen Hilfe der aktuelle Zustand ermittelt werden kann. Die Zustandsübergänge können jeweils autonom oder über den Empfang von *Transitionen* durchgeführt werden. Bei einer Transition handelt es sich um eine Nachricht, die den Zustandsübergang anhand des Ausgangs- und Zielzustandes bestimmt. Diese können bidirektional gesendet, empfangen und jeweils lokal auf dem vorhandenen DEA angewendet werden. Des Weiteren können auch ausgehende Transitionen einen Zustandswechsel auf dem lokalen DEA bewirken. Alle Aktionen, die den Datenaustausch betreffen, werden für den jeweiligen Zustand sowohl client- als auch serverseitig definiert.

6.4.1 Zyklen

Solange eine Verbindung zwischen dem mobilen Client und dem Synchronisationsserver besteht, muss auch das Protokoll kontinuierlich aktiv sein, damit auftretende Ereignisse behandelt werden können. Eine derartige Überwachung kann mit einem zyklischen Wechsel der Zustände erreicht werden, wie in Abbildung 6.5 gezeigt wird.



(a) DEA_{client} **(b)** DEA_{server}

Abbildung 6.5: Zyklischer Zustandsübergang der Automaten; ↗ beschreibt ausgehende und ↘ empfangene Transitionen

Hierbei besitzen sowohl der Automat des mobilen Clients DEA_{client} als auch der des Synchronisationsservers DEA_{server} die Zustände `ACTIVE` und `STANDBY`, in denen sie sich jeweils abwechselnd befinden. Im Zustand `STANDBY` wird ausschließlich der Empfang einer Transition erwartet, die anschließend angewendet wird.

Der wechselseitige Ausschluss ist dadurch gewährleistet, dass die ausgehenden Transitionen ↗`CLIENT_ACK` und ↗`SERVER_ACK` jeweils vor der Übermittlung lokal angewendet werden und somit den Zustandsübergang von `ACTIVE` nach `STANDBY` durchführen. Währenddessen bewirkt dies auf der Gegenseite den Zustandsübergang von `STANDBY` nach `ACTIVE`.

Von diesen zwei Zuständen sind auch andere für die Synchronisation relevanten Zustände gemäß der Definition von Transitionen erreichbar. Im Zustand `ACTIVE` wird die lokale Warteschlange auf Vorhandensein von Aufgaben geprüft. Im positiven Fall wird auf DEA_{client} und DEA_{server} ein Übergang zu einem Zustand durchgeführt, der diese Aufgabe gesondert behandelt. Im Fall, dass keine Aufgaben in der Warteschlange existieren, werden für eine fest definierte Zeit keine Instruktionen durchgeführt. Hiermit wird insbesondere auf dem mobilen Client sowohl die Rechenintensität als auch das zu übertragende Datenvolumen reduziert.

6.4.2 Zustände für primäre Synchronisation

Beim Verbinden eines mobilen Clients mit dem Synchronisationsserver müssen alle bis dahin vorgenommenen Änderungen auf den jeweiligen Datenbanken effizient synchronisiert werden. Bei der primären Synchronisation handelt es sich um den initialen Abgleich der Datensätze in beiden Richtungen. Dieser Abgleich wird in mehrere Schritte unterteilt, wobei jeder Schritt jeweils in einem gesonderten Zustand abgearbeitet wird (siehe Abbildung 6.6).

DEA_{server} ist diesbezüglich analog zu DEA_{client} konstruiert, mit der Ausnahme, dass `STANDBY` der Nachfolgerzustand von `PCP` ist und die für einen Zustandsübergang benötigten Transitionen

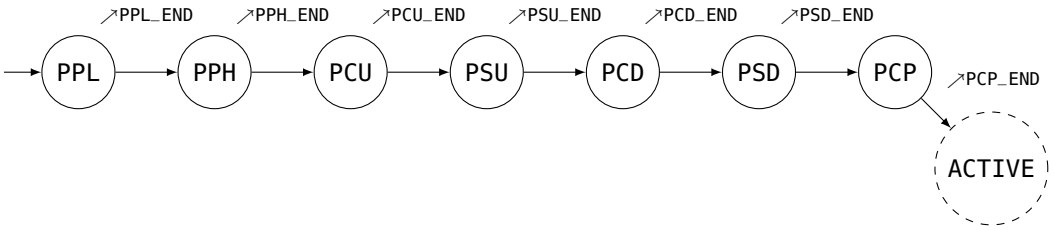


Abbildung 6.6: Sequenz von Zuständen für die primäre Synchronisation (DEA_{client})

empfangen werden. Die Nomenklatur der Zustände für die primäre Synchronisation erfolgt aus der Sicht des mobilen Clients. Die Aktionen, die dabei ausgeführt werden, werden für beide Automaten wie folgt erläutert.

`PPL` (*primary pull*) – Alle Datensätze, die sich nicht in der mobilen Datenbank befinden, jedoch in der gemeinsam genutzten Datenbank vorhanden sind, werden bezogen.

`PPH` (*primary push*) – Alle Datensätze, die sich nicht in der gemeinsam genutzten Datenbank befinden, jedoch in der mobilen Datenbank vorhanden sind, werden gesendet.

`PCU` (*primary client update*) – Alle Datensätze der mobilen Datenbank, dessen Versionsnummer kleiner ist als die der zugehörigen Datensätze aus der gemeinsam genutzten Datenbank, werden überschrieben.

`PSU` (*primary server update*) – Alle Datensätze der gemeinsam genutzten Datenbank, dessen zugehörigen Datensätze auf der mobilen Datenbank modifiziert sind und ihre inkrementierte Versionsnummer um eins höher ist (siehe Abschnitt 6.2), werden zum Synchronisationsserver gesendet und im RDBMS überschrieben.

`PCD` (*primary client delete*) – Alle unmodifizierten Datensätze der mobilen Datenbank, die auf der gemeinsam genutzten Datenbank nicht länger vorhanden sind, werden gelöscht.

`PSD` (*primary server delete*) – Alle Datensätze der gemeinsam genutzten Datenbank, dessen zugehörigen Datensätze auf der mobilen Datenbank gelöscht sind und ihre Versionsnummer übereinstimmt, werden gelöscht.

`PCP` (*primary conflict pull*) – In diesem Zustand werden Datensätze der mobilen Datenbank explizit auf Konflikte überprüft. Für den Fall, dass der zugehörige Datensatz in der gemeinsam genutzten Datenbank nicht länger vorhanden ist, nachdem dieser zuvor synchronisiert wurde, wird ein Präsenzkonflikt für diesen Datensatz ausgelöst. Modifizierte Datensätze aus der mobilen Datenbank, dessen Versionsnummer gleich oder niedriger ist als jene des zugehörigen Datensatzes der gemeinsam genutzten Datenbank, werden entsprechend des Relationsschemas aus Abschnitt 6.1 zur Tabelle der Konflikte hinzugefügt. Hierbei handelt es sich um Änderungskonflikte.

Nach Beendigung von `PCP` werden die Automaten jeweils in den zyklischen Zustandsübergang aus Abschnitt 6.4.1 versetzt.

6.4.3 Zustände für ereignisbasierende Synchronisation

Für den Fall, dass server- oder clientseitig Ereignisse aufgetreten sind und somit Aufgaben in der lokalen Warteschlange vorhanden sind, werden diese in zugehörigen Zuständen abgearbeitet. Diese tragen zur Effizienz der ganzheitlichen Synchronisationsmethode bei, da sich die für eine Synchronisation benötigten Metadaten nur auf einen Datensatz beziehen. Auf dem mobilen Client existieren drei Typen von Aufgaben, die auftreten können und somit das Senden einer zugehörigen Transition an den Synchronisationsserver auslösen (siehe Abbildung 6.7).

Die Transition ↗DATA_ADDED, die von einem mobilen Client gesendet wird, überführt zu beiden Seiten die Automaten in den Zustand IPH (*immediate push*). Daraufhin wird der neue Datensatz an den Server gesendet. Dieser fügt den neuen Datensatz zur gemeinsam genutzten Datenbank hinzu. Unmittelbar danach wird die Transition ↘DATA_ADDED_NOTIFICATION an alle anderen Clients versendet, worauf der neue Datensatz jeweils im Zustand IPL (*immediate pull*) empfangen und zur jeweiligen mobilen Datenbank hinzugefügt wird. Abschließend kehrt DEA_{client} in den Zustand ACTIVE und DEA_{server} in den Zustand STANDBY zurück, um mit dem zyklischen Zustandswechsel fortzufahren.

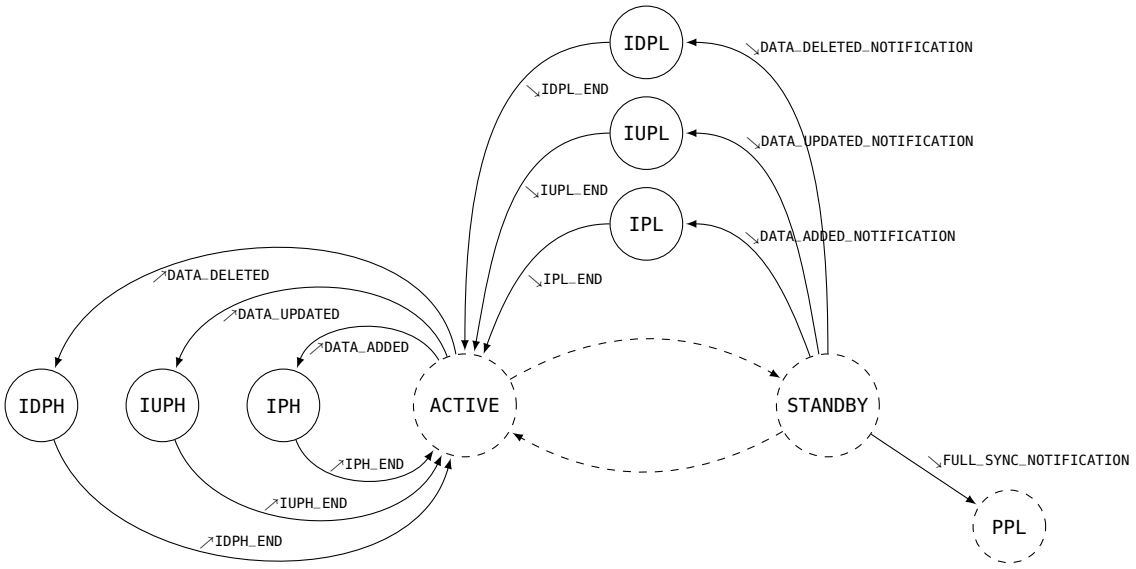


Abbildung 6.7: Zustände für ereignisbasierende Synchronisation (DEA_{client})

Das gleiche Paradigma wird auch mit den Zuständen IUPH (*immediate update push*), IUPL (*immediate update pull*), IDPH (*immediate delete push*) und IDPL (*immediate delete pull*) befolgt, die für eine synchronisierte Modifikation und Löschung von Datensätzen verwendet werden.

Da die Ereignisse jeweils einen Datensatz betreffen, können Sie im Fall einer Neuverbindung eines zusätzlichen Clients nicht dazu verwendet werden, alle dort vorgenommenen Änderungen auf den bereits verbundenen Clients effizient anzuwenden. Daher wird im Zustand PCP (siehe Abschnitt 6.4.2) von DEA_{server} einmalig eine neue Aufgabe für die primäre Synchronisation erstellt, sodass der Empfang der Transition ↘FULL_SYNC_NOTIFICATION auf allen bereits verbundenen

Clients eine primäre Synchronisation beginnend mit dem Zustand PPL auslöst. Alle daraufhin folgenden Modifikationen werden mit Hilfe von Ereignissen abgefangen und in den hier genannten Zuständen synchronisiert.

Weiterhin besteht die Möglichkeit, dass ein Datensatz von mehreren Clients zeitgleich bearbeitet wird oder die ereignisbasierende Synchronisation während der Modifikation eines Datensatzes einsetzt. Um Datenverluste zu vermeiden, werden auch hier Konflikte ausgelöst, wie in den nachfolgenden Abschnitten genauer erläutert wird.

6.5 Datenaufbereitung und Algorithmen

Der in vorangegangenem Abschnitt vorgestellte Synchronisationsautomat organisiert anhand von Zuständen und Transitionen, zu welchem Zeitpunkt Aktionen zur Synchronisation von Datensätzen auszuführen sind. Neben dem Versand von Transitionen werden auch Datensätze und zugehörige Metadaten zwischen dem mobilen Client und dem Synchronisationsserver ausgetauscht. Zu den Metadaten gehören in diesem Kontext der Primärschlüssel und die Versionsnummer eines Datensatzes. Diese tragen erheblich zur Verringerung des versendeten Datenvolumens während des gesamten Synchronisationsvorgangs bei. Des Weiteren können sie jeweils in Form von Vektoren versendet werden, wodurch zusätzlich die Effizienz gesteigert wird. In diesem Abschnitt werden insbesondere korrespondierende Algorithmen beschrieben, die dem Datenaustausch und der Datenspeicherung zugrundeliegen.

6.5.1 Composer und Parser

Da die Kommunikation zwischen Client und Server standardmäßig mit Hilfe von Sockets und in Form von Zeichenströmen realisiert wird, müssen die zu übertragenden Daten dafür angepasst werden. Hierfür werden Konzepte so genannter *Composer* und *Parser* herangezogen. Composer bereiten die Daten in der Weise vor, dass Werte aus Objekten in Zeichenketten umgewandelt und mit Separatoren abgegrenzt werden. Parser besitzen die umgekehrte Funktionalität. Sie extrahieren aus der zusammengesetzten Zeichenkette Werte heraus und sind in der Lage, die durch den Composer verarbeiteten Objekte wiederherzustellen. Abbildung 6.8 zeigt ein Beispiel einer Komposition eines Objekts. Hier bestimmt die Reihenfolge der komponierten Werte die zugehörigen Attribute.

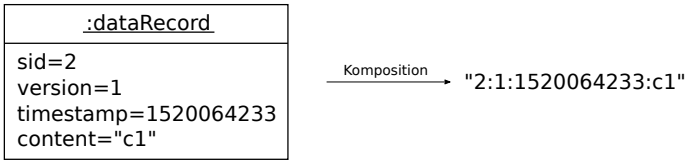


Abbildung 6.8: Beispiel einer Komposition mit dem Zeichen „:“ als Separator

Hierdurch können auch Listen oder *Key-Value*-Paare eines bestimmten Datentyps in Form von Vektoren komponiert und somit unstetige Übertragungen vermieden werden.

6.5.2 Primary Pull

Auf dem mobilen Client wird eine Liste an `SID`-Werten von vorhandenen Datensätzen erstellt und an den Server gesendet. Anschließend werden fehlende Datensätze empfangen und zur Datenbank hinzugefügt, wobei dessen Modus auf *RELEASED* gesetzt wird.

Der Synchronisationsserver erzeugt ebenfalls eine Liste an `SID`-Werten aus der gemeinsam genutzten Datenbank. Von dieser wird jene `SID`-Liste subtrahiert, die vom Client empfangen wurde. Diese Differenz wird verwendet, um Datensätze aus der gemeinsam genutzten Datenbank abzufragen und sie an den Client zu senden.

6.5.3 Primary Push

Alle Datensätze, die sich im Modus *NEW* befinden und einen bereits generierten `CID`-Wert besitzen, werden an den Server gesendet. Anschließend werden `SID-CID`-Wertepaare empfangen, wobei der neue `SID`-Wert anhand von `CID` gesetzt und der Modus auf *RELEASED* aktualisiert wird.

Die auf dem Server empfangenen, neuen Datensätze werden zur gemeinsam genutzten Datenbank hinzugefügt. Die dabei generierten Primärschlüssel `SID` werden mit den zugehörigen `CID`-Werten als Wertepaare zurück an den Client übermittelt.

6.5.4 Primary Client Update

Hierbei werden `SID-VERSION`-Wertepaare aller Datensätze, die sich im Modus *RELEASED* befinden, an den Server gesendet. Anschließend werden Datensätze empfangen und lokal aktualisiert.

Auf dem Server hingegen werden `SID-VERSION`-Wertepaare empfangen und anhand dessen nur Datensätze mit einer höheren Versionsnummer aus der gemeinsam genutzten Datenbank abgefragt. Diese werden schließlich zurück an den Client übermittelt.

6.5.5 Primary Server Update

Auf dem mobilen Client werden alle Datensätze, die sich im Modus *MODIFIED* befinden, an den Server gesendet. Anschließend wird eine `SID`-Liste vom Synchronisationsserver erwartet, woraufhin für jeden Datensatz der jeweiligen `SID` der Modus auf *RELEASED* gesetzt wird.

Die auf dem Synchronisationsserver empfangenen Datensätze werden zur Erneuerung verwendet, im Fall dass dessen Versionsnummer mit der jeweiligen, um eins inkrementierten Versionsnummer des gemeinsam genutzten Datensatzes übereinstimmt. Gleichzeitig wird bei jeder erfolgreichen Aktualisierung eines Datensatzes, dessen `SID` zu einer Liste hinzugefügt, die zurück an den Client gesendet wird. Andernfalls handelt es sich um einen Konflikt, der im dafür vorgesehenen Zustand behandelt wird.

6.5.6 Primary Client Delete

Alle SID-Werte von Datensätzen, die sich im Modus *RELEASED* befinden, werden an den Server gesendet. Daraufhin wird eine weitere SID-Liste empfangen, wobei zugehörige Datensätze aus der mobilen Datenbank entfernt werden.

Die SID-Werte aus der gemeinsam genutzten Datenbank, werden aus der empfangenen SID-Liste des Clients entfernt. Die differenzierte SID-Liste wird zurück an den Client gesendet.

6.5.7 Primary Server Delete

Hierfür werden SID-VERSION-Wertepaare aller Datensätze, die sich im Modus *DELETED* befinden, an den Server gesendet. Anschließend wird eine SID-Liste empfangen und dessen zugehörigen Datensätze aus der mobilen Datenbank endgültig entfernt. Daraufhin wird eine Liste an Datensätzen empfangen, die serverseitig nicht entfernt werden konnten. Dabei werden diese lokal einschließlich der Versionsnummer aktualisiert und dessen Modus auf RELEASED gesetzt.

Serverseitig werden die empfangenen SID-VERSION-Wertepaare für einen Versionsvergleich benutzt. Hier werden ausschließlich Datensätze entfernt und die SID in eine Positivliste aufgenommen, falls die Versionsnummer mit der aus dem empfangenen SID-VERSION-Wertepaar übereinstimmt. Andernfalls wird der Datensatz in eine Negativliste aufgenommen. Sowohl die Positiv- als auch die Negativliste werden an den Client zurückgesendet.

Dieser Mechanismus verhindert Datenverluste, die auftreten können, indem neuere Datensätze anhand von älteren entfernt werden.

6.5.8 Primary Conflict Pull

Es werden SID-VERSION-Wertepaare aller Datensätze, die sich im Modus *MODIFIED* befinden, an den Server gesendet. Daraufhin wird eine Liste an Datensätzen empfangen, die in Änderungskonflikten mit den lokalen Datensätzen stehen. Die betroffenen lokalen Datensätze werden dabei in die Tabelle CLIENT_CONFLICT_RECORD (siehe Abschnitt 6.1) kopiert und durch die gemeinsam genutzte Version überschrieben. Des Weiteren wird eine SID-Liste empfangen, die die zuvor serverseitig entfernten Datensätze repräsentiert. Da es sich hierbei um Präsenzkonflikte handelt, werden die betroffenen Datensätze ebenfalls in die Tabelle CLIENT_CONFLICT_RECORD kopiert und als solche gekennzeichnet.

Der Synchronisationsserver erzeugt SID-VERSION-Wertepaare aller gemeinsam genutzter Datensätze. Anschließend werden auch clientseitige SID-VERSION-Wertepaare empfangen. Die gemeinsam genutzten Datensätze, dessen inkrementierte Versionsnummer höher ist als jene aus den empfangenen Wertepaaren, werden in Form von einer Liste zum Client zurückgesendet. Ebenfalls wird eine SID-Liste an den Client gesendet, dessen zugehörige Datensätze in der gemeinsam genutzten Datenbank nicht mehr vorhanden sind.

6.5.9 Immediate Push

Auf dem mobilen Client wird zunächst der neu hinzugefügte Datensatz aus der Aufgabe extrahiert und an den Synchronisationsserver gesendet. Folglich wird der empfangene `SID`-Wert auch lokal für den neu hinzugefügten Datensatz gesetzt.

Der Synchronisationsserver empfängt den clientseitig neu hinzugefügten Datensatz, fügt diesen zur gemeinsam genutzten Datenbank hinzu und sendet den generierten `SID`-Wert zurück.

6.5.10 Immediate Update Push

Der auf dem Client geänderte Datensatz wird extrahiert und an den Server gesendet. Anschließend können Fälle auftreten, die anhand des Rückgabewertes unterschieden werden können. Im Fall des Rückgabewerts 0, wird der Modus des gesendeten Datensatzes auf *RELEASED* gesetzt. Der Rückgabewert 1 führt zum Empfangen des neueren, gemeinsam genutzten Datensatzes und somit zu einer lokalen Aktualisierung. Der zuvor gesendete Datensatz wird in die Tabelle der Konflikte eingetragen, wobei es sich um einen Änderungskonflikt handelt. Im Fall des Rückgabewerts 2, wird der zuvor gesendete Datensatz in die Tabelle der Konflikte eingetragen, wobei es sich um einen Präsenzkonflikt handelt.

Der Synchronisationsserver empfängt den auf dem Client geänderten Datensatz. Bei Nichtverfügbarkeit des zugehörigen Datensatzes in der gemeinsam genutzten Datenbank, wird der Rückgabewert 2 zurück an den Client gesendet. Sonst wird der Datensatz im Rahmen der Bedingung 6.2 erneuert. Bei einer erfolgreichen Aktualisierung wird der Rückgabewert 0 an den Client gesendet. Andernfalls erfolgt das Senden des Rückgabewerts 1, der einen Änderungskonflikt adressiert, gefolgt von der Übermittlung des neueren, gemeinsam genutzten Datensatzes.

6.5.11 Immediate Delete Push

Zunächst wird auf dem mobilen Client überprüft, ob sich seit der Extraktion des zu löschenden Datensatzes aus der Aufgabe dessen Versionsnummer erhöht hat. Dies kann u. a. durch Immediate Update Pull geschehen. In einem solchen Fall wird der Löschvorgang verweigert. Andernfalls wird der zu löschende Datensatz an den Synchronisationsserver gesendet und anschließend aus der mobilen Datenbank endgültig entfernt.

Auf dem Synchronisationsserver wird der zu löschende Datensatz empfangen und aus der gemeinsam genutzten Datenbank entfernt.

6.5.12 Immediate Pull

Der neue, gemeinsam genutzte Datensatz wird vom Synchronisationsserver bezogen und in die mobile Datenbank eingetragen. Dabei wird dessen Modus auf *RELEASED* gesetzt.

Auf dem Synchronisationsserver wird hingegen der neu hinzugefügte Datensatz aus der entsprechenden Aufgabe entnommen und an den Client gesendet.

6.5.13 Immediate Update Pull

Der mobile Client empfängt den aktualisierten Datensatz und vergleicht dessen Versionsnummer mit der des lokal vorhandenen Datensatzes. Im Fall, dass die Versionsnummer des empfangenen Datensatzes niedriger oder gleich ist, befindet sich der lokale Datensatz im Änderungskonflikt und wird in die entsprechende Tabelle eingetragen. Andernfalls wird der lokale Datensatz aktualisiert und dessen Modus auf *RELEASED* gesetzt.

Der Synchronisationsserver extrahiert den aktualisierten Datensatz aus der Aufgabe und sendet diesen an den Client.

6.5.14 Immediate Delete Pull

Der mobile Client empfängt den zu löschenden Datensatz und überprüft, ob dieser bereits in der Tabelle `CLIENT_CONFLICT_RECORD` vorhanden ist. Im positiven Fall wird keine weitere Aktion unternommen, da die Auflösung des bestehenden Konflikts noch ansteht. Andernfalls wird die Versionsnummer des lokal vorhandenen Datensatzes mit der des empfangenen Datensatzes verglichen. Falls die Letztere niedriger ist, wird ein neuer Präsenzkonflikt eingetragen, da der Datensatz lokal geändert, serverseitig jedoch gelöscht wurde. Ansonsten wird auch die lokal vorhandene Variante des Datensatzes entfernt.

Der Synchronisationsserver hingegen extrahiert lediglich den gelöschten Datensatz aus der Aufgabe und sendet diesen an den Client.

6.6 Konfliktauflösung

Trotz der hohen Transparenz, die das SSPMD bietet, ist die Interaktion des Benutzers in bestimmten Fällen unvermeidbar. Dies betrifft insbesondere die Auflösung von bestehenden Konflikten. Dadurch dass die behandelten Daten nicht an eine bestimmten Syntax und Semantik gekoppelt sind, kann keine allgemeine Strategie zur autonomen Konfliktauflösung entwickelt werden. Daher ist der Eingriff des Benutzers essentiell für die Konfliktauflösung.

Im Fall eines Änderungskonflikts wird der betroffene Datensatz `A` aus der Tabelle `CLIENT_DATA_RECORDS` und `A'` aus `CLIENT_CONFLICT_RECORD` jeweils geeignet visualisiert. Der Benutzer hat anschließend die Möglichkeit, den Konflikt aufzulösen, indem `A` oder `A'` bearbeitet und zur Auflösung ausgewählt werden. Der hieraus entstandene Datensatz `A"` wird gemäß Bedingung 6.3 versioniert.

$$\texttt{A".VERSION} = max(\texttt{A.VERSION},\ \texttt{A'.VERSION}) + 1 \tag{6.3}$$

Anschließend wird `A'` aus `CLIENT_CONFLICT_RECORD` entfernt und `A` durch `A"` in der mobilen Datenbank ersetzt, wobei dessen Modus auf *MODIFIED* gesetzt wird. Hierauf wird die in Abschnitt 6.5.10 beschriebene Aktion Immediate Update Push ausgeführt und somit eine neue Änderung simuliert.

Im Fall eines Präsenzkonflikts werden dem Benutzer zwei Optionen angeboten. Diese sind zum Einen das Löschen und zum Anderen das Beibehalten des betroffenen Datensatzes. Letzteres entfernt den betroffenen Datensatz aus der Tabelle `CLIENT_CONFLICT_RECORD` und fügt ihn zur Tabelle `CLIENT_DATA_RECORDS` hinzu, wobei dessen Modus auf *NEW* gesetzt wird. Bei bestehender Verbindung zum Synchronisationsserver wird schließlich die Aktion Immediate Push (siehe Abschnitt 6.5.9) ausgelöst.

7 Implementierung

Der in der vorliegenden Arbeit vorgestellte Mechanismus zum Abgleich mobiler Daten mit zentralisierten, gemeinsam genutzten Datenbanken ist unabhängig vom zugrundeliegendem Betreibs- und Datenbankmanagementsystem. Nachfolgend werden Ansätze zur Implementierung eines client- und serverseitigen Prototyps beschrieben, wodurch u. a. das vorgestellte Synchronisationsprotokoll SSPMD umgesetzt wird. Hierbei handelt es sich um die mobile Anwendung *Collective Medical Data Assistant* (COMEDA) und dem Synchronisationsserver namens *RDBSync*, der in Form eines Dienstes umgesetzt wird.

COMEDA ermöglicht die gemeinsame Nutzung und Bearbeitung von medizinischen Dokumenten. Dabei entspricht jedes Dokument jeweils einem Datensatz aus der gemeinsam genutzten Datenbank, die mittels RDBSync zusammen mit den Datenbanken anderer mobiler Clients synchronisiert wird.

Die Abschnitte 7.1 bis 7.3 stellen zunächst die in der Implementierung verwendeten Bibliotheken sowie das Entwurfsmuster und die Verwendung der Nebenläufigkeit (engl. *Multithreading*) vor. Abschnitt 7.4 beschreibt die Architektur der client- und serverseitigen Anwendung, während Abschnitt 7.5 die konkreten Details zur Implementierung liefert.

7.1 Umgebung und Bibliotheken

Als Zielumgebung für die clientseitige Implementierung wird die Android-Plattform gewählt. Diese ist weitestgehend geräteunabhängig und bietet eine Vielzahl an Bibliotheken, mitunter Schnittstellen für die Interaktion mit dem lokalen Datenbankmanagementsystem. *SQLite* gehört zu den bekanntesten Datenbanksystemen für Android. In Verbindung mit *SQLiteOpenHelper*, der zugehörigen Programmierschnittstelle für Java, wird diese Komponente hier ebenfalls verwendet.

Der Synchronisationsserver wird in Form eines eigenständigen Dienstes unter Verwendung der Programmiersprache Java implementiert. Die Wahl des gemeinsam genutzten Datenbanksystems ist in diesem Kontext irrelevant, da die dafür benötigten Treiber lediglich geladen werden, während die allgemeingültige Schnittstelle `java.sql` für die SQL-Abfragen benutzt wird.

Für die sichere Kommunikation wird die abstrakte Klasse `javax.net.ssl.SSLSocket` beiderseits entsprechend implementiert. Das Observer-Pattern für die Ereignisse wird mittels *Greenrobot*[1] auf dem Android-Client und mittels *Google Guava*[2] auf dem Synchronisationsserver umgesetzt.

[1] https://github.com/greenrobot/EventBus
[2] https://github.com/google/guava

7.2 Entwurfsmuster

Der Entwurf der mobilen Anwendung stützt sich auf ein modifiziertes *Model-View-Presenter*-Pattern (MVP). Die Komponente *View* beinhaltet dabei alle Klassen, die für die Visualisierung und Benutzerinteraktion zuständig sind. *Model* repräsentiert Daten, die für die temporäre Übertragung zwischen *View* und *Presenter* verwendet werden, selbst jedoch keine aktive Funktionalität besitzen. Letztlich bildet der *Presenter* den Kern der Anwendungsarchitektur, da hier alle logischen Prozesse, die die Synchronisation und die persistente Datenhaltung betreffen, durchgeführt werden. *View* ist in der Lage neben der Erlangung von darzustellenden Daten auch selbst Observer-Ereignisse zu empfangen, woraufhin die Aktualisierung der Ansicht in Echtzeit erfolgt. Die Modifikation besteht darin, dass bestimmte Klassen aus *View* an ein Datenmodell gekoppelt sind, welches dort modifiziert und an den *Presenter* überreicht wird. Abbildung 7.1 zeigt das hier verwendete MVP-Entwurfsmuster.

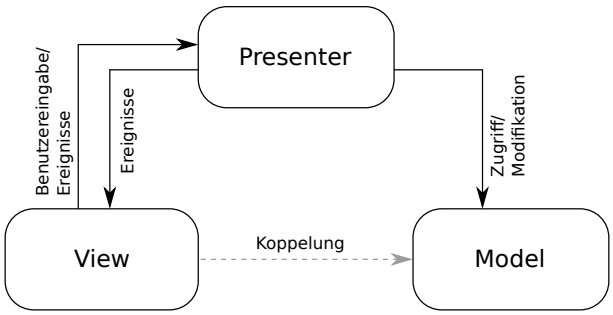


Abbildung 7.1: MVP-Entwurfsmuster des mobilen Clients

Der Synchronisationsserver hingegen weist ein einfacheres Entwurfsmuster auf. Da es sich um einen Dienst handelt und die Interaktion mit dem Benutzer nicht vorhanden ist, entfällt die Komponente *View*. Dadurch ergibt sich ein Entwurfsmuster, welches lediglich aus *Controller*- und *Model*-Komponenten besteht, wobei *Controller* die vollständige Steuerung über *Model* übernimmt.

7.3 Multithreading

Bei diesem Mechanismus wird die Synchronisation sowohl client- als auch serverseitig in einem eigenen Thread ausgeführt. Auf der mobilen App ist dadurch gewährleistet, dass die Datenbank durch den Benutzer zeitlich unabhängig von der Synchronisation bearbeitet werden kann. Der Synchronisationsserver besitzt einen primären Thread, der ausschließlich eingehende Verbindungsanfragen bearbeitet. Bei jeder erfolgreichen Verbindung wird diese an einen neu erstellten Thread für die Synchronisation weitergereicht. Somit besitzt jeder verbundene Client einen eigenen Thread auf dem Synchronisationsserver mit dem Nachrichten ausgetauscht werden. Dadurch dass diese Threads jeweils in der Lage sind auf eine gemeinsam genutzte Datenbank zuzugreifen, muss dieser Zugriff sequenziell erfolgen.

7.4 Anwendungsarchitektur

Die Klassen der mobilen Anwendung sind in mehrere Pakete unterteilt, die jeweils einer Komponente aus dem Entwurfsmuster zugeordnet werden können (siehe Abbildung 7.2). Das Paket *sync* enthält alle relevanten Klassen, die für den Hintergrundservice und den Datenabgleich zwischen der lokalen und der gemeinsam genutzten Datenbank benötigt werden. Hierzu gehören Anbindungen an das mobile RDBMS, die Überwachung und Steuerung der Warteschlange für eintretende Ereignisse sowie Algorithmen für die effiziente Synchronisation. Das Paket *automaton* enthält die Implementierung eines DEA. Anhand dessen werden Folgen von Zuständen bestimmt, die zur Abstimmung der durchzuführenden Operationen mit dem Synchronisationsserver dienen. Im Paket *stream* befinden sich Klassen, die ausschließlich der sicheren Kommunikation mit dem Server dienen. Hierdurch wird ein sicherer Eingabe- und Ausgabekanal zur Verfügung gestellt und eventuelle Fehler bei der Datenübertragung erkannt. Die Bestandteile der Pakete *activities* und *view* werden hier aufgrund ihrer niedrigen inhaltlichen Relevanz bewusst unterschlagen. Das Paket *util* besitzt Klassen, die die Serialisierung und das Parsen von Daten ermöglichen und sie somit für die Datenübertragung vorbereiten. Die Basis für den Datenaustausch zwischen internen Instanzen, wie beispielsweise der Schnittstelle für die mobile Datenbank und dem Dienst für die Synchronisation, bilden die Klassen aus dem Paket *model*.

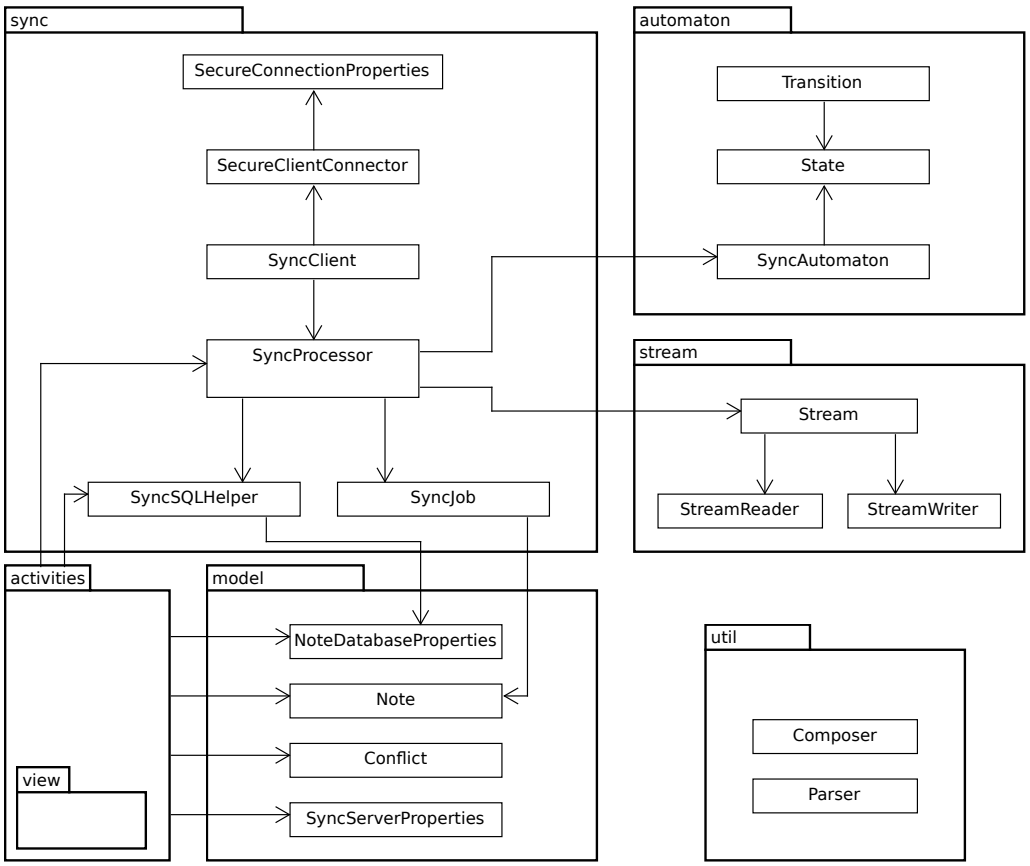


Abbildung 7.2: Vereinfachtes UML-Klassendiagramm der Android-Anwendung (COMEDA)

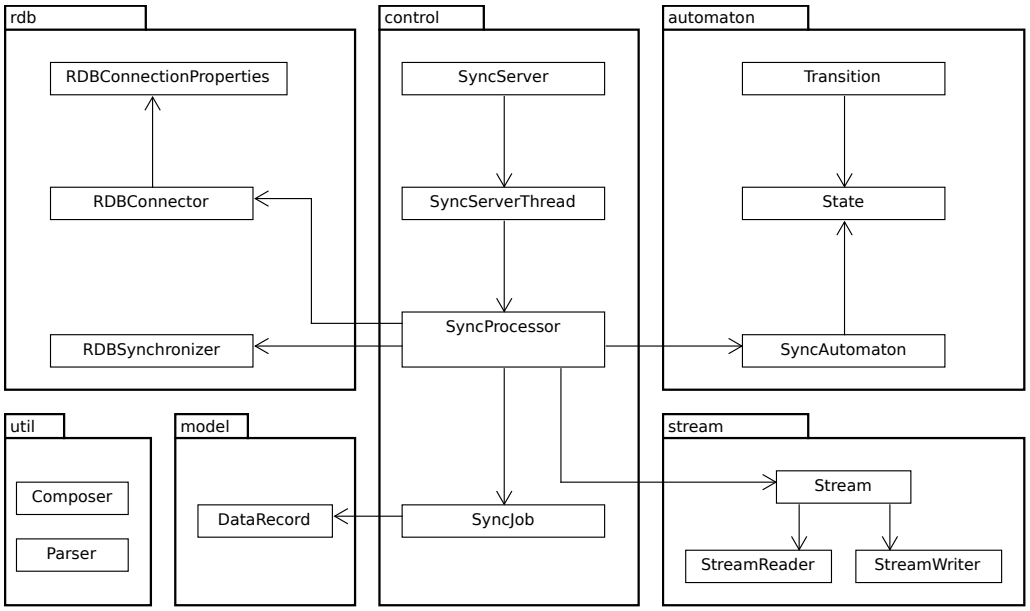


Abbildung 7.3: Vereinfachtes UML-Klassendiagramm des Synchronisationsservers (RDBSync)

Eine große Ähnlichkeit weist auch die Paketstruktur des Synchronisationsservers auf (siehe Abbildung 7.3). Die Pakete *stream* und *util* sind äquivalent zu den gleichnamigen Paketen aus der mobilen Anwendung, während *automaton* kleinere semantische Unterschiede aufweist. Diese Symmetrie und die damit einhergehende Wiederverwendbarkeit reduzieren den Aufwand für die Implementierung des gesamten Systems. Im Paket *sync* geschieht die jeweilige Zuordnung der neu erstellten Threads zu den verbundenen Clients und anschließend die Durchführung des in dieser Arbeit vorgestellten Synchronisationsprotokolls. Ebenso werden hier Aufgaben, die ihren Ursprung bei konkurrierenden Threads haben, mit Hilfe von Warteschlangen verwaltet. Das Paket *rdb* bietet alle Schnittstellen zur Durchführung von Datenbankabfragen am gemeinsam genutzten RDBMS, während *model* jene Klasse beinhaltet, die das zu synchronisierende Datum repräsentiert.

7.5 Implementierungsmethode

Im Folgenden werden Ansätze zur konkreten Implementierung der mobilen Anwendung und des Synchronisationsservers behandelt. Diese umfassen insbesondere die Umsetzung von SSPMD in Verbindung mit auftretenden Ereignissen und die Behandlung von potentiell auftretenden Konflikten.

7.5.1 Activities

Eine *Activitiy* ist unter Android der Zusammenschluss einer Klasse und eines Layouts, aus denen eine Bildschirmseite generiert wird. Die Activitiy-Klasse ist für die dahinterstehende Funktionalität der Bildschirmseite zuständig während das Layout im XML-Format dessen Aussehen und die Position der anzuzeigenden Elemente bestimmt. Die Klasse `MainActivity` bildet im konkreten Fall den Programmeinstiegspunkt für den mobilen Client. Diese zeigt u. a. alle Dokument-Datenbanken an, die vom Benutzer hinterlegt wurden. Über die hierarchische Beziehungen zueinander können von hier aus andere Acitivities erreicht werden. Diese werden wie folgt erläutert.

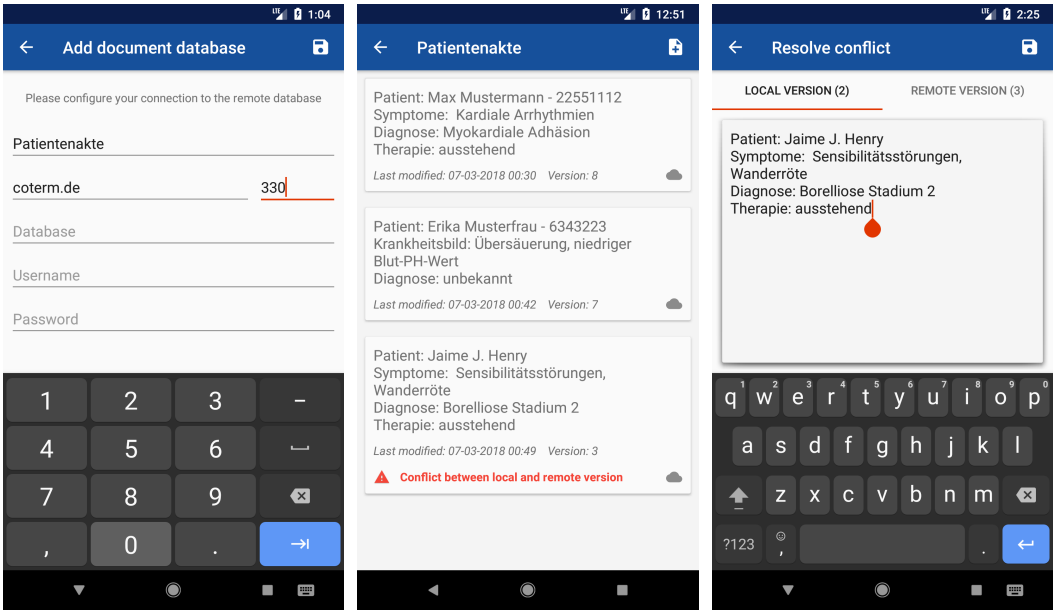
`MainSettingsActivity` – Hier werden allgemeine Einstellungen verwaltet. Dazu gehören auch Verbindungsdaten zum Synchronisationsserver.

`ShowDocumentDatabasePropertiesActivity` – Mit Hilfe dieser Activitiy können neue Dokument-Datenbanken hinzugefügt oder die bestehenden geändert werden (siehe Abbildung 7.4a). Hier wird der Ort der gemeinsam genutzten Datenbank sowie die dafür benötigten Daten zur Authentifizierung bestimmt.

`ShowDocumentDatabaseActivity` – Diese Activitiy zeigt alle Datensätze der spezifischen, gemeinsam genutzten Datenbank an (siehe Abbildung 7.4b). Hier werden die Dokumente in Form von Karten dargestellt, worunter auch Metadaten wie der Versionsstand oder der Bearbeitungszeitpunkt angezeigt werden. Im Fall von Konflikten wird auch eine entsprechende Warnmeldung an die visualisierte Karte angehangen.

`ShowDocumentActivity` – Hier kann das zu synchronisierende Dokument vom Benutzer vollständig eingesehen und bearbeitet werden.

`ShowConflictActivity` – In dieser Activity werden bestehende Konflikte gelöst. Diese besitzt zwei Ansichten; jeweils für die lokale und die gemeinsam genutzte Version eines Dokuments. Es können gewünschte Modifikationen an beiden Versionen vorgenommen werden, woraufhin der Inhalt einer Ansicht für die Auflösung benutzt wird (siehe Abbildung 7.4c).



(a) `ShowDocumentDatabase-PropertiesActivity` **(b)** `ShowDocumentDatabase-Activity` **(c)** `ShowConflictActivity`

Abbildung 7.4: Screenshots verfügbarer Activities in der mobilen Anwendung (COMEDA)

7.5.2 SyncClient

Die Klasse `SyncClient` bildet den Hintergrundservice in der Android-Anwendung. Sie leitet sich von der Klasse `java.lang.Thread` ab und benötigt einen `SecureClientConnector` mit den zugehörigen Eigenschaften für die Verbindung zum Synchronisationsserver (siehe Abbildung 7.2). Dieser Thread wird bei der Erstellung von `ShowDocumentDatabaseActivity` gestartet und bleibt bis zum Verlassen dieser Activity aktiv. Zuvor wird eine neue Instanz der Klasse `SyncClientProcessor` erzeugt und die sichere Verbindung zum Synchronisationsserver aufgebaut. Anschließend wird die Methode `cycle()` zyklisch aufgerufen, solange das Protokoll aktiv ist und der Benutzer die beinhaltende Activity nicht geschlossen hat (siehe Quelltext 7.1). Des Weiteren können hier Verbindungsabbrüche behandelt werden, indem wiederkehrend die allgemeine Verfügbarkeit der Netzwerkverbindung geprüft und anschließend ein neuer Verbindungsversuch gestartet wird. Ebenfalls können hier die von `cycle()` verursachten Fehler behandelt werden, die die Ausführung des Protokolls verhindern.

Quelltext 7.1 Grundaufbau der `run()`-Methode von `SyncClient`

```
SSLSocket sslSocket = secureClientConnector.getSslSocket();
Stream stream = new Stream(
      new StreamReader(sslSocket.getInputStream()),
      new StreamWriter(sslSocket.getOutputStream()));
syncClientProcessor.setStream(stream);

while (syncClientProcessor.isEnabled() && !isInterrupted()) {
   syncClientProcessor.cycle();
}
```

7.5.3 SyncAutomaton (Client)

`SyncAutomaton` implementiert in Verbindung mit `State` und `Transition` das theoretische Konstrukt eines DEA. Es beinhaltet lediglich ein Attribut, welches den aktuellen Zustand des Automaten festhält und eine Methode, die eine gegebene Transition am aktuellen Zustand anwendet und somit einen Zustandsübergang bewirkt (siehe Quelltext 7.2).

Quelltext 7.2 Klasse `SyncAutomaton` in vereinfachter Darstellung

```
public class SyncAutomaton {
   private State currentState;
   public SyncAutomaton() {
      currentState = State.START;
   }
   public void apply(Transition transition) { /* ... */ }
   public State getCurrentState() { return currentState; }
}
```

Bei `State` handelt es sich um eine Aufzählung aller möglichen Zustände. `Transition` hingegen ist eine parametrisierte Aufzählung, dessen Elemente jeweils den Ursprungs- und Zielzustand als Parameter erhalten (siehe Quelltext 7.3). Anhand dieser Zuordnung wird der in Abschnitt 6.4 vorgestellte, deterministische Automat umgesetzt.

Quelltext 7.3 Aufzählung `Transition` in vereinfachter Darstellung

```
public enum Transition {
   /* ... */
   SERVER_ACK(State.STANDBY, State.ACTIVE),
   CLIENT_ACK(State.ACTIVE, State.STANDBY);

   private State origin;
   private State target;

   public Transition(State o, State t) { origin = o; target = t; }
   public State getOrigin() { return origin; }
   public State getTarget() { return target; }
}
```

7.5.4 SyncClientProcessor

Diese Klasse bildet den Kern des gesamten Systems zur Synchronisation, da sie mit allen Komponenten für die Durchführung des Protokolls, die lokale Datenhaltung und die Netzwerk-Kommunikation interagiert (siehe Abbildung 7.5). Außerdem implementiert sie alle in Abschnitt 6.5 beschriebenen Algorithmen.

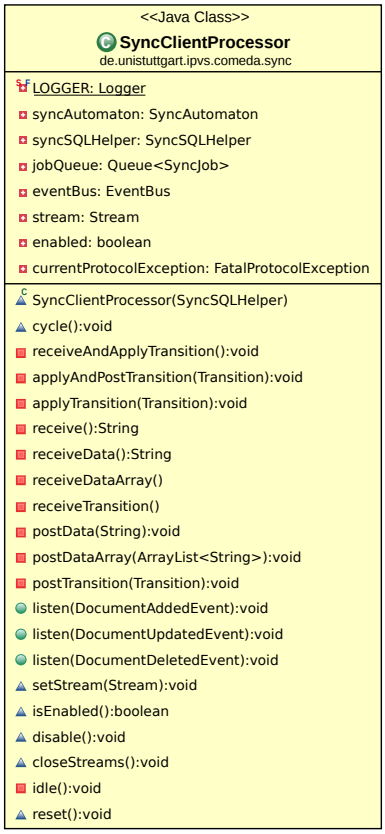


Abbildung 7.5: Vollständiges UML-Klassendiagramm von `SyncClientProcessor`

Bei der Konstruktion wird zunächst eine neue Instanz des `SyncAutomaton` und eine Thread-sichere Queue erzeugt. Des Weiteren wird der systemweit verwendete `EventBus` erlangt und die als Parameter erhaltene Schnittstelle zur Datenbank `SyncSQLHelper` gesetzt.

Zu Beginn der Methode `cycle` wird der Zustand, in dem sich `SyncAutomaton` befindet, ermittelt. Anschließend verhält sich diese Methode Abhängig vom aktuellen Zustand (siehe Quelltext 7.4). Mit den Methoden `postTransition` und `receiveAndApplyTransition` können Transitionen an den Synchronisationsserver gesendet bzw. empfangen und angewendet werden. So sendet der Client beispielsweise im Startzustand eine Verbindungsanfrage an den Server. Daraufhin erfolgt beiderseits ein Übergang zum Zustand `CONNECTING`, bei dem der Synchronisationsserver alle Verbindungsdaten zum gemeinsam genutzten RDBMS erwartet, um mit der primären Synchronisation beginnen zu können.

Quelltext 7.4 Grundaufbau der Methode `cycle()`

```
void cycle() {
   State currentState = syncAutomaton.getCurrentState();
   if (currentState == State.START) {
      postTransition(Transition.CLIENT_RDB_CONNECT_REQ);
      receiveAndApplyTransition();

   } else if (currentState == State.CONNECTING) { /* ... */
```

Einen Bestandteil davon bildet der in Abschnitt 6.5.4 vorgestellte Zustand `PRIMARY_CLIENT_UPDATE`. Dessen client- und serverseitige Implementierung wird in Quelltexte 7.5 und 7.6 dargestellt. Mit den Methoden `postData`, `receiveData`, `postDataArray` und `receiveDataArray` werden komponierte Daten versendet und zeitgleich auf der Gegenseite empfangen, wo sie schließlich dekomponiert, d. h. in Objekte umgewandelt werden. Die zuordnenden Zahlenmarkierungen am Ende der betroffenen Codezeile verdeutlichen diese Korrespondenz.

Quelltext 7.5 `PRIMARY_CLIENT_UPDATE`-Implementierung in `SyncClientProcessor`

```
} else if (currentState == State.PRIMARY_CLIENT_UPDATE) {
   Map<Integer, Integer> sidVersionMap = syncSQLHelper.getLocalSidVersionMap(Mode.RELEASED);
   postData(Composer.composeIntMap(sidVersionMap)); ①
   ArrayList<Note> notes = Parser.getParsedNoteArray(receiveDataArray()); ②
   for (Note note : notes) {
      syncSQLHelper.updateNoteBySid(note, Mode.RELEASED);
   }
   receiveAndApplyTransition(); ③
}
```

Quelltext 7.6 `PRIMARY_CLIENT_UPDATE`-Implementierung in `SyncServerProcessor` zum Vergleich

```
} else if (currentState == State.PRIMARY_CLIENT_UPDATE) {
   Map<Integer, Integer> clientSidVersionMap = Parser.parseIntMap(receiveData()); ①
   ArrayList<DataRecord> newerRecords = rdbSQLHelper.getNewerRecords(clientSidVersionMap);
   postDataArray(Composer.getComposedDataRecordArray(newerRecords)); ②
   applyAndPostTransition(Transition.CLIENT_PRIMARY_UPDATE_COMPLETE); ③
}
```

Mit Hilfe des `EventBus` können Ereignisse, d. h. Objekte an alle registrierten Instanzen gesendet werden, indem dort speziell annotierte Methoden autonom aufgerufen werden, die diese Ereignisse als Parameter empfangen. Auf diese Weise werden die `listen`-Methoden dieser Klasse aufgerufen, wenn ein Dokument hinzugefügt, verändert oder gelöscht wurde. Auslöser hierfür sind die jeweiligen Activities. Ein Ereignis beinhaltet die Aktion und das von dieser Aktion betroffene Dokument. Im Quelltext 7.7 wird eine derartige Methode gezeigt, die jene Ereignisse abfängt, die durch das Hinzufügen eines Dokuments ausgelöst werden. Diese überführt das jeweilige Ereignis in ein `SyncJob`, welches schließlich in die lokale Warteschlange `jobQueue` eingereiht

wird. Ein `SyncJob` beinhaltet ebenfalls die durchzuführende Operation und das davon betroffene Dokument.

Quelltext 7.7 `PRIMARY_CLIENT_UPDATE`-Implementierung in `SyncServerProcessor` zum Vergleich

```
@Subscribe
public void listen(NoteAddedEvent event) {
   jobQueue.offer(new SyncJob(event.getNote(), SyncJob.Operation.ADD));
}
```

Im Zustand `ACTIVE` beginnt die Abarbeitung der `jobQueue`, falls diese `SyncJob`-Objekte bereithält. Anschließend wird die Operation festegestellt und anhand dessen der zugehörige Zustandsübergang sowohl client- als auch serverseitig durchgeführt. Quelltext 7.8 zeigt eine mögliche Implementierung, bei der Ereignis des Hinzufügens eines neuen Dokuments verarbeitet wird. Dabei führt die Transition `DATA_ADDED` zum Zustand *Immediate Push* (siehe Abschnitt 6.5.9). Im Fall dass keine `SyncJob`-Objekte vorhanden sind, werden Zyklen ausgelöst (siehe Abschnitt 6.4.1).

Quelltext 7.8 Behandlung von Ereignissen aus der Warteschlange

```
} else if (currentState == State.STANDBY) {
   receiveAndApplyTransition();

} else if (currentState == State.ACTIVE) {
   SyncJob syncJob = jobQueue.peek();
   if (syncJob == null) {
      idle(); // no instructions for a defined time
      applyAndPostTransition(Transition.CLIENT_ACK);

   } else if (syncJob.getOperation() == Operation.ADD) {
      applyAndPostTransition(Transition.DATA_ADDED);
   } /* ... */
}
```

Bei Bedarf kann derselbe `EventBus` verwendet werden, um eine Aktualisierung der Ansicht zu bewirken, wenn Änderungen an der lokalen Datenbank durchgeführt wurden. Damit kann auf die manuelle Aktualisierung durch Benutzer verzichtet werden, wodurch die Nutzerfreundlichkeit der mobilen Anwendung gesteigert wird.

Im Fall eines Verbindungsabbruchs wird `SyncAutomaton` in den Startzustand versetzt und die Warteschlange `jobQueue` geleert. Folglich kann die Verbindung wiederhergestellt und das Protokoll wiederaufgenommen werden. Für den Fehlerfall, bei dem das Protokoll nicht weiter fortgesetzt werden kann (z. B. durch Nichtverfügbarkeit der gemeinsam genutzten Datenbank), kann ein zusätzlicher Zustand eingeführt werden, wobei die Fehlermeldung vom Synchronisationsserver zum Client übermittelt und dort dem Benutzer schließlich angezeigt wird.

7.5.5 SyncServer und SyncServerThread

Die Instanz dieser Klasse bildet den Hintergrundservice des Synchronisationsservers. Hier werden eingehende Verbindungsanfragen des `SyncClient` entgegengenommen. Bei einem erfolgreichen Verbindungsaufbau liefert `java.net.Socket`, die für die `Stream`-Instanz benötigten Ein- und Ausgabekanäle für sämtliche Datenübertragungen.

`SyncServer` ist in der Lage mehrere `SyncServerThread`-Objekte unterzubringen, die jeweils einer aktiven Netzwerkverbindung zum `SyncClient` zugeordnet sind. Hierzu wird die Klasse `java.util.concurrent.ExecutorService` für die parallele Programmierung verwendet. Mit dieser kann u. a. das Beenden von Threads organisiert werden, dem Verbindungsfehler oder Herunterfahren des Hintergrundservices zugrunde liegt.

`SyncServerThread` besitzt genau ein `SyncServerProcessor`-Objekt, dessen Methode `cycle()` analog zur clientseitigen Implementierung zyklisch ausgeführt wird. Dies geschieht solange das Protokoll aktiv ist, d. h. keine Verletzungen dessen aufgetreten sind.

7.5.6 SyncServerProcessor

In dieser Klasse wird das serverseitige SSPMD implementiert. Dessen Instanz interagiert über ein `Stream`-Objekt mit `SyncClientProcessor` auf der Gegenseite. Sie verfügt ebenfalls über eine Methode `cycle()`, die kontinuierlich aufgerufen wird und sich abhängig vom aktuellen Zustand des `SyncAutomaton` verhält. In dieser werden alle serverseitigen Algorithmen implementiert, die durch SSPMD für den jeweiligen Zustand vorgegeben werden.

Da es mehrere, parallel laufende Instanzen von `SyncServerProcessor` geben kann, ist eine interne Kommunikation zwischen den Threads unerlässlich, um die ereignisbasierende Synchronisation (Abschnitt 6.4.3) zwischen den mobilen Clients zu ermöglichen. Diese Kommunikation wird auch hier mit Hilfe des Observer-Patterns und der Klasse `EventBus` realisiert. Speziell annotierte Methoden werden aufgerufen, sobald ein Ereignisobjekt von einer anderen `SyncClientProcessor`-Instanz über den `EventBus` gesendet wurde. Ein Beispiel für eine solche Methode zeigt das Quelltext 7.9.

Quelltext 7.9 Behandlung von Ereignissen aus der Warteschlange

```
@Subscribe
public void listen(DataRecordUpdatedNotificationEvent event) {
   if (!event.getIdentifier().equals(identifier)) {
      SyncJob syncJob = new SyncJob(event.getDataRecord(), SyncJob.Operation.UPDATE);
      jobQueue.offer(syncJob);
   }
}
```

Ähnlich wie beim mobilen Client wird auch hier das als Parameter empfangene Ereignisobjekt in eine `SyncJob`-Instanz umgewandelt, die schließlich in eine Warteschlange eingereiht wird. Um wiederholte Ausführungen zu vermeiden, wird mit Hilfe des Attributs `identifier` zwischen den Sendern und den Empfängern von Ereignissen unterschieden.

Die Warteschlange `jobQueue` wird auch hier im Zustand `ACTIVE` abgearbeitet. Abhängig von `SyncJob.Operation` wird ein Übergang zum Zustand durchgeführt, in dem eine Änderung an den mobilen Client übermittelt wird. Dies betrifft insbesondere `IPL`, `IUPL` und `IDPL`.

8 Evaluation

In folgendem Kapitel wird sowohl eine anwendungsbezogene als auch eine technische Evaluation des vorgestellten Konzepts für die Synchronisation von mobilen Daten durchgeführt und so die Grundlage für den Vergleich mit anderen Ansätzen geschaffen. Es werden auch weitere Strategien erörtert, um die Stabilität von SSPMD und die Verfügbarkeit der gemeinsam genutzten Daten zu verbessern.

Abschnitt 8.1 stellt zunächst die Umgebung für die technische Evaluation vor. Anschließend wird in Abschnitt 8.2 eine Fallstudie durchgeführt, gefolgt von einer Effizienzanalyse (Abschnitt 8.3) und einer Diskussion über Erweiterungs- und Optimierungsmöglichkeiten von SSPMD (Abschnitt 8.4). Schließlich wird SSPMD den Ansätzen aus verwandten Arbeiten gegenübergestellt (Abschnitt 8.5).

8.1 Umgebung

Für die experimentelle Bewertung wird eine Architektur herangezogen, bei der zwei Android-Smartphones mit einem lokalen Netzwerk drahtlos verbunden sind (siehe Abbildung 8.1). Im selben Netzwerk befindet sich auch ein stationärer Rechner, der sowohl den Synchronisationsserver als auch ein RDBMS ausführt. Tabelle 8.1 stellt die Spezifikationen der an dieser Evaluation beteiligten Systeme dar.

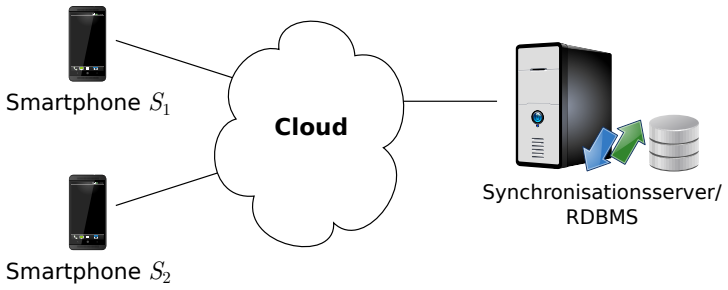


Abbildung 8.1: Cloud-Architektur für die Evaluation

	Smartphone S_1	Smartphone S_2	Sync.-Server/RDBMS
Bezeichnung	LG Nexus 5X	Huawei Honor 6 Plus	–
Betriebssystem	Android 8.1.0	Android 5.1.1	Debian GNU/Linux 9.4
Prozessor	4x1.4 + 2x1.8 GHz	4x1.8 + 4x1.3 GHz	8x3.6 GHz
RAM	2GB	3GB	8GB
Netzwerkanbindung	WLAN, 50 Mbit/s	WLAN, 50 Mbit/s	LAN, 100 Mbit/s
Anwendung	COMEDA[1]	COMEDA[1]	RDBSync[1]
Datenbanksystem	SQLite	SQLite	MariaDB[2]

Tabelle 8.1: Spezifikation der Evaluationsumgebung

8.2 Fallstudie

Nachfolgend wird SSPMD in kritischen Fällen untersucht, bei denen es ohne entsprechender Schutzmechanismen zu Datenverlust kommen würde. Dafür wird auf dem Smartphone S_1 und S_2 jeweils COMEDA als Client verwendet, der mit dem Synchronisationsserver RDBSync kommuniziert.

Fall 1

Versuch. S_1 ist online und modifiziert einen bestehenden Datensatz. S_2 ist offline und löscht denselben Datensatz. Anschließend geht S_2 online und beginnt mit der Synchronisation.

Ergebnis. S_2 bezieht die aktuellere Version desselben Datensatzes. Ein Datenverlust wird verhindert, da die Löschung durch den Synchronisationsserver ignoriert wird. Die von S_1 durchgeführte Modifikation bleibt bestehen und wird nicht auf Basis eines veralteten Datensatzes durch S_2 in der gemeinsam genutzten Datenbank gelöscht.

Fall 2

Versuch. S_1 und S_2 sind offline und modifizieren denselben Datensatz. Daraufhin geht S_1 gefolgt von S_2 online und beginnt mit der Synchronisation.

Ergebnis. Auf S_2 wird ein Änderungskonflikt für den modifizierten Datensatz ausgelöst (siehe Abbildung 8.2). Dieser muss anschließend durch den Benutzer gelöst werden, woraufhin die Auflösung des Konflikts in Form einer neuen Änderung S_1 erreicht.

[1] Ein im Rahmen dieser Arbeit entwickelter Prototyp mit der Implementierung von SSPMD
[2] Ein MySQL-basierendes Open-Source-Datenbankmanagementsystem – https://mariadb.org/

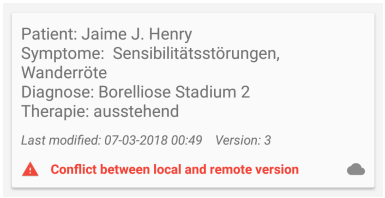


Abbildung 8.2: Darstellung eines Änderungskonflikts

Fall 3

Versuch. S_1 ist online und löscht einen bestehenden Datensatz. S_2 ist offline und modifiziert den von S_1 gelöschten Datensatz. Anschließend geht S_2 online und startet die Synchronisation.

Ergebnis. Auf S_2 wird ein Präsenzkonflikt für den modifizierten Datensatz ausgelöst (siehe Abbildung 8.3). Dieser muss anschließend durch den Benutzer gelöst werden, indem der modifizierte Datensatz erneut hinzugefügt, d. h. beibehalten oder verworfen wird.

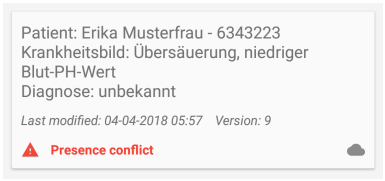


Abbildung 8.3: Darstellung eines Präsenzkonflikts

Fall 4

Versuch. S_1 ist online und modifiziert einen bestehenden Datensatz. S_2 ist ebenso online und löscht den von S_1 bearbeiteten Datensatz.

Ergebnis. Der Benutzer wird durch eine Meldung sofort über einen Präsenzkonflikt informiert, der wie im vorhergehenden Fall gelöst wird (siehe Abbildung 8.4).

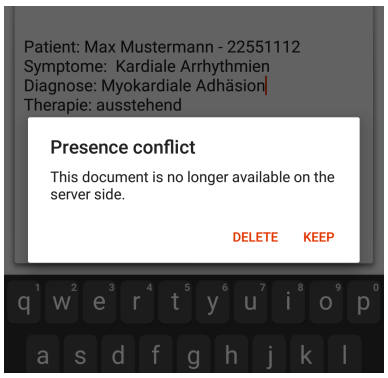


Abbildung 8.4: Sofortige Benachrichtigung über ein Präsenzkonflikt

8.3 Effizienzanalyse

Nachfolgend wird der zeitliche Aufwand und die für eine vollständige Synchronisation benötigte Datenmenge analysiert. Neben der theoretischen Analyse wird SSPMD in einer geeigneten technischen Umgebung auch experimentell evaluiert.

8.3.1 Datenverkehr

Im Verlauf des Protokolls werden die Daten in komponierter Form zwischen dem mobilen Client und dem Synchronisationsserver bidirektional übertragen. Dazu gehören u. a. komponierte Vektoren, Key-Value-Paare, Transitionen oder die Datensätze selbst.
Die Vektoren beinhalten in diesem Kontext überwiegend eine Sequenz von Primärschlüssel-Werten. Dessen Verwendung trägt insofern zur Effizienz bei, dass die Semantik auch ohne ganzheitlicher Übertragung von Datensätzen übermittelt werden kann. So können beispielsweise fehlende oder die zu löschenden Datensätze effizient adressiert werden. Im Fall von Primärschlüsseln, bestehen die Vektoren aus den einzelnen Zahlenwerten, die jeweils durch ein Separator-Zeichen getrennt sind.
Unter der Annahme, dass ein komponierter Vektor eines Primärschlüssels aus einer Zahlensequenz von 1 bis n besteht, kann dessen Größe in Bytes nach Gleichung (8.1) bestimmt werden.

$$d_{vector}(n) = b \cdot \sum_{i=1}^{n} 1 + \lceil log_{10}(i) \rceil \tag{8.1}$$

b bezeichnet hierbei die Anzahl an Bytes, die für die Darstellung eines Zeichens benötigt wird. So weist eine Sequenz für $n = 10$ unter Verwendung von Java ($b = 2$) eine Größe von 42 Bytes auf.
Key-Value-Paare weisen einem Schlüssel einen weiteren, beliebigen Wert zu. So werden im Verlauf des Protokolls beispielsweise Primärschlüssel-Version-Paare an den Server übermittelt, um Redundanz bei der Übertragung von Datensätzen zu vermeiden; d. h. aktuelle Datensätze werden nicht erneut synchronisiert.
Sei m der maximale Zahlenwert aus n Key-Value-Paaren. Die Größe einer komponierten Key-Value-Map lässt sich mittels Gleichung (8.2) nach oben abschätzen. Hierbei werden auch Separator-Zeichen, die den Wert vom Schlüssel trennen, ebenfalls beachtet.

$$d_{map}(n, m) = d_{vector}(n) + n \cdot b \cdot (1 + \lceil log_{10}(m) \rceil) \tag{8.2}$$

Abbildung 8.5 zeigt ein Beispiel einer Primärschlüssel-Version-Map, die im vorgestellten Synchronisationsprotokoll verwendet wird. Durch das Separator-Zeichen „`:`“ werden die einzelnen Paare getrennt, während „`;`“ den Wert vom Schlüssel innerhalb eines Paares abgrenzt.

```
1;1:2;10:3;22
```

Abbildung 8.5: Beispiel einer Primärschlüssel-Version-Map

Für das genannte Beispiel ($n = 3, m = 22$) beträgt d_{map} unter Java ($b = 2$) höchstens 30 Bytes.
Die Transitionen besitzen eine unveränderliche Größe. Die komponierten Datensätze hingegen sind benutzerabhängig, wodurch dessen Größe allgemeingültig nicht abgeschätzt werden kann.

In Abschnitt 6.4.2 wurde bereits die primäre Synchronisation ausführlich vorgestellt. Diese bildet den Kern des gesamten Protokolls, da hiermit eine ganzheitliche Synchronisation ohne Behandlung von Ereignissen erreicht werden kann. Da die primäre Synchronisation in mehreren diskreten Zuständen durchgeführt wird, werden in dessen Abhängigkeit auch Daten (z. B. komponierte Vektoren) zwischen dem Client und dem Server ausgetauscht. Tabelle 8.2 veranschaulicht und beschreibt jene Daten, die aus der Sicht des Clients gesendet und empfangen werden.

Primary-Zustand	R	Datum	Beschreibung
Pull (PPL)	↗	`SID`-Vektor	Alle lokal vorhandenen Primärschlüssel-Werte von gemeinsam genutzten Datensätzen
	↘	Fehlende Datensätze (Array)	Datensätze, die in der mobilen Datenbank nicht vorhanden sind
Push (PPH)	↗	Neue Datensätze (Array)	Nicht synchronisierte Datensätze, die zur mobilen Datenbank neu hinzugefügt wurden und sich im Modus *NEW* befinden
	↘	`SID-CID`-Map	Wertepaare für die Zuweisung der gemeinsam genutzten Primärschlüssel-Werte (`SID`) zu den Datensätzen der mobilen Datenbank mit den jeweiligen, lokalen Primärschlüssel-Werten (`CID`)
Client Update (PCU)	↗	`SID-VERSION`-Map	Wertepaare des gemeinsam genutzten Primärschlüssels (`SID`) und der Version jeweiliger, lokaler Datensätze, die sich im Modus *RELEASED* befinden
	↘	Neuere Datensätze (Array)	Vergleichsweise aktuellere Datensätze aus der gemeinsam genutzten Datenbank
Server Update (PSU)	↗	Modifizierte Datensätze (Array)	Datensätze, die in der mobilen Datenbank abgeändert wurden und sich somit im Modus *MODIFIED* befinden
	↘	`SID`-Vektor	Gemeinsam genutzten Primärschlüssel-Werte jener Datensätze, die serverseitig erfolgreich aktualisiert wurden
Client Delete (PCD)	↗	`SID`-Vektor	Gemeinsam genutzten Primärschlüssel-Werte jener Datensätze aus der mobilen Datenbank, die sich im Modus *RELEASED* befinden
	↘	`SID`-Vektor	Gemeinsam genutzten Primärschlüssel-Werte jener Datensätze, die aus der mobilen Datenbank gelöscht werden

(R) Richtung der Synchronisation: (↗) Senden; (↘) Empfangen

Primary-Zustand	R	Datum	Beschreibung
Server Delete (PSD)	↗	`SID-VERSION`-Map	Wertepaare des gemeinsam genutzten Primärschlüssels (`SID`) und der Version jeweiliger, lokaler Datensätze, die sich im Modus *DELETED* befinden
	↘	`SID`-Vektor	Gemeinsam genutzten Primärschlüssel-Werte jener Datensätze, die serverseitig gelöscht wurden und somit auch aus der mobilen Datenbank entfernt werden
	↘	Neuere Datensätze (Array)	Vergleichsweise aktuellere Datensätze aus der gemeinsam genutzten Datenbank, die aufgrund der Versionsunterschiede nicht gelöscht werden konnten und daher in der mobilen Datenbank aktualisiert werden
Conflict Pull (PCP)	↗	`SID-VERSION`-Map	Wertepaare des gemeinsam genutzten Primärschlüssels (`SID`) und der Version jeweiliger, lokaler Datensätze, die sich im Modus *MODIFIED* befinden
	↘	In Änderungs-konflikt stehende Datensätze (Array)	Gemeinsam genutzten Datensätze, die jeweils mit der lokalen Version in Änderungskonflikt stehen und entsprechend behandelt werden
	↘	`SID`-Vektor	Gemeinsam genutzten Primärschlüssel-Werte jener Datensätze, die serverseitig nicht länger verfügbar sind und somit lokal einen Präsenzkonflikt auslösen

(R) Richtung der Synchronisation: (↗) Senden; (↘) Empfangen

Tabelle 8.2: Zu übertragende Daten pro Zustand

8.3.2 Diskretisierung

Aus Tabelle 8.2 lässt sich nun extrahieren, inwiefern der Modus die zu übertragenden Datenmengen diskretisiert und somit zur Reduktion des benötigten Datenvolumens beiträgt.

	NEW	*RELEASED*	*MODIFIED*	*DELETED*	*OFFLINE*
Pull (PPL)	×	×	×	×	
Push (PPH)	×				
Client Update (PCU)		×			
Server Update (PSU)			×		
Client Delete (PCD)		×			
Client Server (PSD)				×	
Conflict Pull (PCP)			×		

Tabelle 8.3: Diskretisierung der mobilen Daten

Tabelle 8.3 ordnet für jeden Primary-Zustand die benötigten Datensätze bzw. Attributwerte eines bestimmten Modus zu. Offensichtlich benötigt nicht jeder Zustand Attributwerte von allen Datensätzen aus der Relation `CLIENT_DATA_RECORDS` der mobilen Datenbank. Andererseits existieren Modi, die von mehreren Zuständen erfasst werden. Dies führt zwar zu einer geringen Redundanz, ist jedoch durch die diskrete Aufgabenverteilung in verschiedenen Zuständen und durch die Unabhängigkeit der Zustände gerechtfertigt. Datensätze, die sich im Modus *OFFLINE* befinden, werden in keinem Zustand verwendet, da sie von der Synchronisation ausgeschlossen sind.

8.3.3 Algorithmische Laufzeit

Hiermit wird im Allgemeinen die asymptotische Laufzeit eines Algorithmus', der von einer bestimmten Eingabegröße abhängt, beurteilt. SSPMD führt für den jeweiligen Zustand sowohl client- als auch serverseitig korrespondierende Algorithmen aus, die miteinander die in Abschnitt 8.3.1 vorgestellten Daten austauschen. Die Eingabegröße wird hierbei durch die Anzahl der zu synchronisierenden Datensätze vorgegeben. Für die Laufzeitanalyse muss mitunter auch die Laufzeit, die für eine Datenbankabfrage benötigt wird, berücksichtigt werden. Da die Algorithmen korrespondieren, fällt auch eine Laufzeit für die Datenübertragung an. Diese wird hier jedoch unterschlagen, da es sich hierbei um eine Laufzeitanalyse handelt, die lediglich auf den Rechnern stattfindet.
Sei n die Anzahl an Datensätzen in der mobilen Datenbank und m die Anzahl jener in der gemeinsam genutzten Datenbank. Tabelle 8.4 zeigt die Worst-Case-Laufzeiten, die sowohl client- als auch serverseitig für die primäre und die eventbasierende Synchronisation auftreten können. Hieraus ist ersichtlich, dass die primäre Synchronisation eine deutlich höhere Zeitkomplexität im Gegensatz zur eventbasierenden Variante aufweist. Auf dem mobilen Client ist dies u. a. auf die Primärschlüssel-Vergleiche im RDBMS, die für mehrere empfangene Datensätze durchgeführt werden müssen, zurückzuführen. Der Synchronisationsserver hingegen muss vor der Anwendung von Änderungen oder Löschungen, diese auf Konflikte überprüfen, was sich auf eine erhöhte Laufzeit im Vergleich zum Client auswirkt.

	Mobiler Client	Synchronisationsserver
Primäre Synchronisation	$\mathcal{O}(n^2)$	$\mathcal{O}(n \cdot m^2)$
Eventbasierende Synchronisation	$\mathcal{O}(n)$	$\mathcal{O}(m)$

Tabelle 8.4: Asymptotische Worst-Case-Laufzeiten von SSPMD

Die eventbasierende Synchronisation beinhaltet Synchronisationseinheiten, die lediglich einen Datensatz betreffen. Daher benötigen sie eine lineare Zeit für die Ausführung und verbessern somit die zeitliche Effizienz des gesamten Systems.

8.3.4 Vorbereitung der Messung

Nachfolgend wird die Effizienz von SSPMD experimentell bewertet. Das Ziel ist die Messung der Zeit, die in verschiedenen Fällen der Synchronisation benötigt wird. Dies betrifft die initiale und die post-initiale Synchronisation eines mobilen Clients mit dem zentralen RDBMS, sowie die ereignisbasierende Synchronisation, mit der eine Änderung sofort an alle verbundenen Geräte verteilt wird.
Dabei wird in der gemeinsam genutzten Datenbank zunächst eine entsprechende Anzahl an Test-Datensätzen abgelegt, woraufhin SSPMD auf S_1 und S_2 gestartet wird (1). Anschließend wird die Netzwerkverbindung von S_1 und S_2 getrennt und mittels S_2 eine Veränderung an 50% der Datensätze vorgenommen. Daraufhin wird die Netzwerkverbindung auf S_2 wieder aktiviert. Nach dem Abgleich der Änderungen seitens S_2 wird nun auch auf S_1 die Verbindung wiederhergestellt und die Synchronisation wiederaufgenommen (2). Des Weiteren werden die beiden mobilen Geräte offline gestellt, wobei beidseitig Änderungen an selben Datensätzen vorgenommen werden, um Änderungskonflikte zu verursachen. Auch werden für einen weiteren Fall Datensätze gleichermaßen gelöscht. Anschließend erfolgt die erneute Aufnahme der Synchronisation (3, 4). Für die Messpunkte (1 bis 4) wird jeweils die für die primäre Synchronisation verstrichene Zeit an S_1 gemessen. Zuletzt wird noch die ereignisbasierende Synchronisation evaluiert, indem auf S_1 ein Datensatz jeweils hinzugefügt, geändert und gelöscht und die Zeitspanne bis zur Anwendung der Änderung auf S_2 bestimmt wird. Hierbei wird die Leerlaufzeit des Zustandes `ACTIVE` stets vernachlässigt.
Die Größe eines zufälligen Datensatzes wird über das Attribut `CONTENT` bestimmt, da dies auch beim produktiven Einsatz dieses Verfahrens der Fall ist. Sie beträgt für alle Test-Datensätze 10 Kilobytes. Die Messungen werden dreifach wiederholt, damit ausreißende Ergebniswerte, die durch Übertragungsstörungen oder das lokale Prozess-Scheduling bedingt sind, ausgeschlossen werden können. Dabei bildet der Durchschnitt den endgültigen Wert einer Messreihe. Die für die Android-Anwendung COMEDA vorhandene Cache wurde nach der jeweiligen Messung nicht zurückgesetzt.

8.3.5 Ergebnisse

In Abbildung 8.6a sind die Zeiten, die für Primary Pull (Messpunkt 1) benötigt werden, dargestellt. Dies betrifft die Synchronisation neuer Datensätze, die auf S_1 bislang nicht verfügbar waren. Die mittlere Verarbeitungsgeschwindigkeit beträgt 10,8 Datensätze pro Sekunde, wobei diese durch die Anzahl an Datensätzen nicht signifikant beeinflusst wird.

Abbildung 8.6b zeigt die Dauer für das beziehen von Änderungen (Primary Client Update, Messpunkt 2), die an der Hälfte der verfügbaren Datensätze durchgeführt wurden. Offensichtlich überschreitet die Laufzeit ab 400 Datensätzen jene, die für Primary Pull benötigt wird. Dies ist darauf zurückzuführen, dass die Datensätze vor der Aktualisierung in der mobilen Datenbank zusätzlich alloziert werden müssen. Die mittlere Verarbeitungsgeschwindigkeit beträgt 10,7 Datensätze pro Sekunde.
Abbildung 8.6c repräsentiert den Messpunkt 3 und gibt die Zeit an, die für die Entfernung nicht mehr vorhandener, gemeinsam genutzter Datensätze benötigt wird. Die Werte fallen vergleichsweise sehr niedrig aus, da lediglich ein empfangener Primärschlüssel-Vektor für die Löschung verarbeitet wird und die Laufzeit somit überwiegend auf das mobile Datenbanksystem abfällt. Die mittlere Verarbeitungsgeschwindigkeit beträgt 39,4 Datensätze pro Sekunde.
Schließlich wird in Abbildung 8.6d der Zeitbedarf von Primary Conflict Pull dargestellt. Dieser gehört mit einer mittleren Verarbeitungsgeschwindigkeit von 8,9 Datensätzen pro Sekunde zur langsamsten Prozedur während der Ausführung des gesamten Protokolls. Neben dem Beziehen von aktuellen Datensätzen, müssen jene, die sich lokal befinden und in Konflikt stehen, zu einer weiteren Tabelle kopiert werden, damit die Möglichkeit einer Konfliktauflösung weiterhin gegeben ist. Hiermit erhöht sich auch die Rechenintensität auf dem mobilen Gerät.

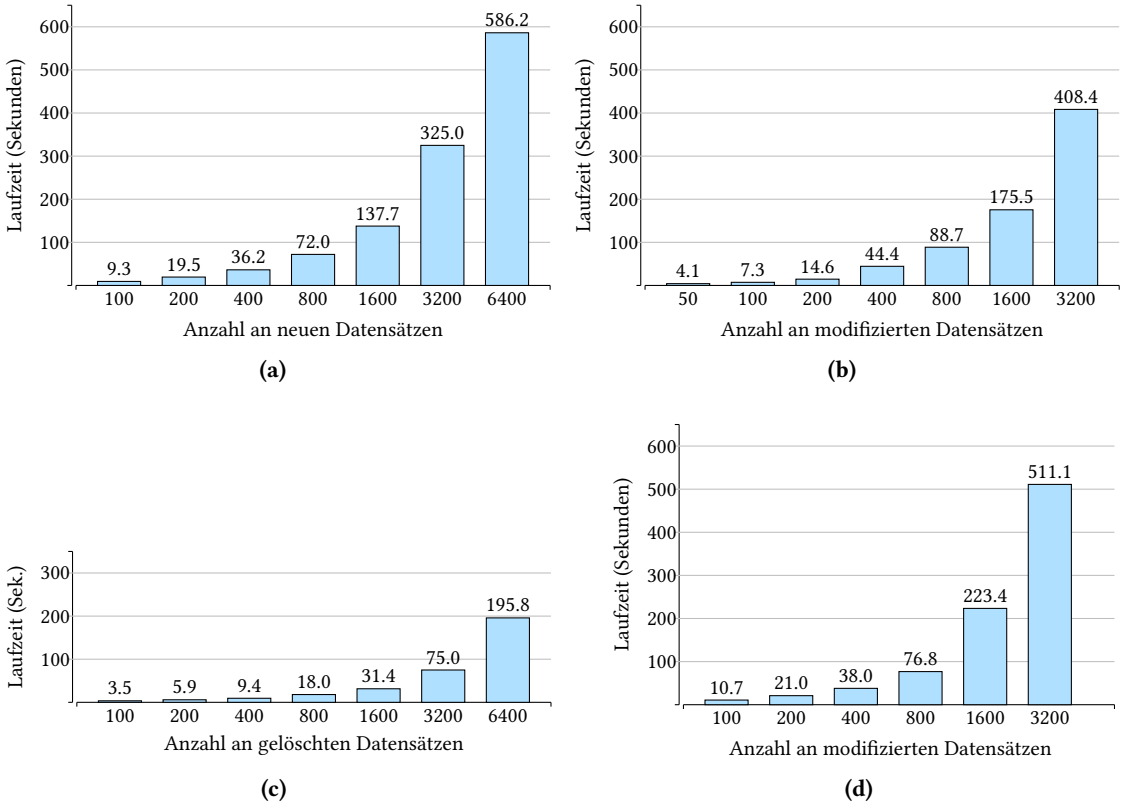


Abbildung 8.6: Ergebnisse der Zeitmessungen

In Tabelle 8.5 werden noch die Zeiten aufgeführt, die für die eventbasierende Synchronisation benötigt werden, bis eine Aktion auf S_2 angewendet wird. Auch hier beträgt die Größe des

betroffenen Datensatzes 10 Kilobyte. Dieser wird auf S_1 jeweils hinzugefügt, geändert und entfernt, womit die entsprechenden Events ausgelöst werden.

	Hinzufügen	Änderung	Entfernen
Laufzeit (Millisekunden)	395	398	334

Tabelle 8.5: Benötigte Zeiten für die eventbasierende Synchronisation

Die eventbasierende Synchronisation ist effizienter im Vergleich zur Durchführung der primären Synchronisation als Ganzes, um lediglich einen Datensatz abzugleichen. Jedoch lassen sich Serien von Änderungen hiermit nicht effizient realisieren, weshalb diese nur bei aktiver Verbindung zum Synchronisationsserver verwendet wird.

8.4 Diskussion

Das in dieser Arbeit vorgestellte Verfahren zur Synchronisation von mobilen Datenbanken eignet sich insbesondere für Daten, die nach dem Änderungszeitpunkt möglichst schnell und ohne weiterer Interaktion des Benutzers abgeglichen werden. Dies schließt jedoch eine Implementierung des Protokolls in Form eines Dienstes ein, da es im ruhenden Fall einen zyklischen Zustandsübergang vollzieht, um beispielsweise auf neue Ereignisse reagieren zu können. Die Dauer eines Leerlauf-Zyklus kann auch durch den Benutzer clientseitig festgelegt werden. Je höher diese ist, desto ressourcenschonender ist der Dienst. Weiterhin ist für die Synchronisation eine aktive Netzwerkverbindung erforderlich, während sie zum Zeitpunkt der Bearbeitung einer mobilen Datenbank auch ausbleiben kann. Dies ist mitunter ein Bestandteil der Transparenz gegenüber dem Benutzer, während die Anzahl der Fälle für einen administrativen Eingriff (z. B. Konfliktauflösung) minimal gehalten wird.
Die zugrundeliegende Netzwerkarchitektur, in der das SSPMD zum Einsatz kommt, bedarf neben den Clients auch einen Synchronisationsserver und ein gemeinsam genutztes RDBMS. Um die Verfügbarkeit respektive Ausfallsicherheit zu erhöhen, können auch mehrere, verteilte Datenbanksysteme zum Einsatz kommen. Für deren Replikation können Verfahren der horizontalen, vertikalen oder hybriden Daten-Fragmentierung verwendet werden [Rah94]. Ebenso lassen sich auch mehrere Synchronisationsserver betreiben, die entsprechend der Priorisierung in den mobilen Clients angesprochen werden. Im Fall, dass der Synchronisationsserver mit der höchsten Priorität nicht erreichbar ist, wird ein Verbindungsversuch zum Synchronisationsserver der nächst niedrigen Priorität unternommen, wie in Abbildung 8.7 veranschaulicht wird.
Dadurch dass der Synchronisationsserver für jeden verbundenen Client einen separaten Thread für die Durchführung des Protokolls zur Verfügung stellt, kann dieser bei einer besonders hohen Anzahl an Clients sowohl die Speicher- als auch die Recheneinheit stark beanspruchen. Um die Wahrscheinlichkeit von Systemausfällen zu verringern, kann die serverseitige Zeitspanne für einen Leerlauf-Zyklus entsprechend der Anzahl an verbundenen Client dynamisch angepasst werden. Dabei erhöht der Server weiter die Leerlaufzeit, je mehr Clients sich verbinden und das Protokoll ausführen. Bei stark beanspruchten Synchronisationsservern können auch spezielle Wartezustände eingeführt werden, die die Clients für eine gewisse Zeit von der Synchronisation ausschließen.
Der in Abschnitt 6.4 behandelte Synchronisationsautomat bildet den Kern für die zustandsbasierende Synchronisation. Abhängig vom Zustand werden sowohl client- als auch serverseitig

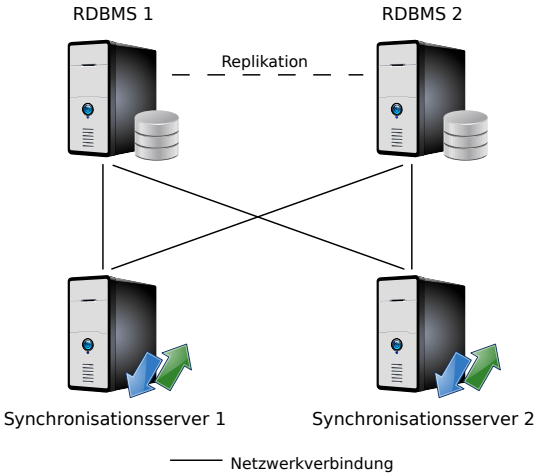


Abbildung 8.7: Redundanz beteiligter Systeme für die Synchronisation

aufeinander abgestimmte Algorithmen ausgeführt, die miteinander korrespondieren, d. h. Daten austauschen. Die Möglichkeit der Wiederverwendung von Zuständen bringt den Vorteil mit sich, dass das Protokoll in bestimmten Situationen nicht erneut ausgeführt werden muss, wie dies beispielsweise bei HTTP der Fall ist. Die Abwärtskompatibilität von SSPMD lässt sich einfach realisieren, indem mehrere Pfade für die jeweilige Version des Protokolls von einem Zustand aus abgezweigt werden. Damit wird nicht nur ein Freiraum für weitere Optimierungen geschaffen, sondern auch die Möglichkeiten gegeben, zwischen verschiedenen mobilen Geräten zu unterscheiden. So können beispielsweise Smartwatches aufgrund ihrer vergleichsweise stark eingeschränkten Rechenkapazität, eine ressourcenschonendere Version des Protokolls ausführen.

8.5 Gegenüberstellung zu bestehenden Ansätzen

Choi et al. [CCP+10] stellen ein Verfahren (SAMD) vor, das mit Hilfe von Hashwerten den Änderungsstatus aller Datensätze überwacht. Der Vorteil von SSPMD gegenüber SAMD besteht in der kompakteren Datenstruktur zur Feststellung von Veränderungen. Eine *MD5*-Hash-Zeichenkette besitzt eine feste Zeichenlänge, die für die Darstellung unter Java 64 Bytes an Speicherplatz benötigt. Der benötigte Speicherplatz eines Primärschlüssel-Version-Paares wird hingegen durch die Zeichenlänge des Primärschlüssels und der zugehörigen Versionsnummer bestimmt. Bei 1000 Datensätzen, die jeweils 1000 Mal abgeändert wurden, weist die Primärschlüssel-Version-Map nach Gleichung (8.2) eine Größe von 11,9 Kilobytes auf. Eine reine Liste an *MD5*-Hash-Zeichenketten benötigt für die genannten Datensätze bereits 64 Kilobytes, wobei Primärschlüssel und Separator-Zeichen hierbei vernachlässigt wurden. Dieser Unterschied macht sich insbesondere bei mobilen Endgeräten mit einer geringen Bandbreite der Netzwerkverbindung bemerkbar. Mit Versionsnummern kann auch die hierarchische Abhängigkeit von Änderungen an einem Datensatz nachvollzogen werden, während Hashwerte in diesem Kontext lediglich eine Änderung signalisieren. Gegenüber SAMD bietet SSPMD den Vorteil, dass keine Berechnung von solchen Hashwerten durchgeführt, sondern lediglich die zugehörige Versionszahl inkrementiert werden muss.

Der MRDMS-Algorithmus von Sethia et al. [SMC+14] konzentriert sich auf eine effiziente Synchronisation von Tabellen unter Benutzung von Zeitstempeln. Dabei wird für jede Daten-Zelle der zugehörige Zeitstempel in einer weiteren Tabelle an derselben Position gespeichert und bei jeder Änderung aktualisiert. Diese wird an den Server gesendet, woraufhin veränderte Zellen identifiziert und beiderseits abgeglichen werden. Unter Verwendung von Zeitstempeln ergibt sich der Nachteil, dass sich eine Vorschrift für die Überschreibung eines älteren Datensatzes nicht konstruieren lässt, womit ein Lost-Update vermieden werden kann. Der Informationsgehalt eines Zeitstempels wird somit lediglich für den Ausschluss bereits synchronisierter Daten-Zellen und somit zur Steigerung der Effizienz genutzt. Die in Abschnitt 6.2 vorgestellten Bedingungen aus SSPMD stellen hingegen sicher, dass eine Änderung ausschließlich auf der aktuellsten Version eines Datensatzes angewendet werden kann, wodurch das Problem eines Lost-Updates vollständig ausgeschlossen wird.

In Bezug auf die Anforderungen aus Abschnitt 3.2 stellt Tabelle 8.6 die drei Ansätze für die Synchronisation von mobilen Datenbanken gegenüber.

	SAMD	MRDMS	SSPMD
A1 Zuverlässigkeit	✗	✗	✓
A2 Effizienz	✗	✓	✓
A3 Verfügbarkeit	✓	✓	✓
A4 Transparenz	✗	✗	(✓)
A5 Datensicherheit	(✓)	(✓)	(✓)
A6 Skalierbarkeit	✗	✗	✓

Tabelle 8.6: Gegenüberstellung von SAMD, MRDMS und SSPMD

Aufgrund fehlender bzw. rudimentärer Mechanismen, die insbesondere bei Datenkonflikten einen Datenverlust nicht verhindern können, erfüllen SAMD und MRDMS nicht die Anforderung der Zuverlässigkeit. Die mobile Datenbank kann hingegen unter Verwendung jeder der vorgestellten Ansätze unabhängig vom Netzwerkstatus bearbeitet werden. MRDMS und SSPMD zeichnen sich insbesondere durch ihre Effizienz aus. SSPMD ist für Hintergrunddienste optimiert und bewahrt den Benutzer weitestgehend vor zusätzlichen Aktionen, die die Synchronisation betreffen. Eine geringfügige Ausnahme bilden hier jedoch Änderungs- und Präsenzkonflikte, wodurch die Transparenz auf Kosten der Zuverlässigkeit geschwächt wird. Bei den restlichen Ansätzen müssen die Synchronisationsvorgänge manuell eingeleitet werden, was zur Nichterfüllung der Transparenz führt. Die Datensicherheit ist bei allen Ansätzen gewährleistet, sofern die Datenübertragung verschlüsselt geschieht und die Daten in der mobilen Datenbank sicher aufbewahrt werden können [SM15]. Der zyklische Zustandsübergang (siehe Abschnitt 6.4.1) in SSPMD ermöglicht es, Leerlaufzeiten während der Synchronisation dynamisch zu bestimmen und somit das Synchronisationsverfahren im Rahmen der zur Verfügung gestellten Ressourcen beliebig skalierbar zu machen.

9 Zusammenfassung und Ausblick

Mobile Datenbanksysteme gehören mitunter zur Grundlage für die strukturierte Datenhaltung in zahlreichen mobilen Anwendungen. Mit zunehmender Vernetzung steigt auch der Bedarf an Möglichkeiten, bestimmte Daten mit anderen Personen nicht nur zu teilen, sondern diese auch gemeinsam zu bearbeiten. Die hierfür erforderlichen Datenbanksysteme, die die gemeinsam genutzten Daten aufbewahren sollten, sind jedoch von mobilen Endgeräten aus nicht direkt anzusprechen, da diese beschränkte Ressourcen und eine diskontinuierliche Netzwerkverbindung aufweisen [SSPE04]. Doch unabhängig hiervon wird die kollaborative Datenbearbeitung von Problemen wie Lost-Updates oder Datenkonflikten stets begleitet.
Für die Synchronisation von mobilen Datenbanken wurden in jüngster Zeit zahlreiche Konzepte eingeführt, die sich der Bildung von Hashwerten, dem Vergleich von Zeitstempeln oder der Erstellung von Snapshots bedienen. Jedoch sind diese Vorgehensweisen für eine kollaborative Datenbearbeitung weitestgehend ungeeignet und unzuverlässig.
Die vorliegende Arbeit stellt eine Architektur vor, bei der ein Synchronisationsserver die Schnittstelle zwischen den mobilen Clients und der gemeinsam genutzten Datenbank bildet. Des Weiteren wird ein zustandsbasierendes Synchronisationsprotokoll vorgestellt, welches sowohl client- als auch serverseitig implementiert wird und eine effiziente Synchronisation erreicht. Dabei werden Lost-Updates unter Zuhilfenahme von einer Versionierungsstrategie präemptiv verhindert und bei konkurrierender Modifikation von Daten entsprechende Maßnahmen ergriffen. Um die beschränkten Ressourcen eines mobilen Gerätes zu schonen, beherbergt das Synchronisationsverfahren einen ereignisbasierenden Datenabgleich. Folglich können Änderungen über den Synchronisationsserver an alle verbundenen Clients zeitnah und energieschonend verteilt werden. Die Datensicherheit wird während der gesamten Synchronisation stets in den Vordergrund gestellt.
Die Umsetzbarkeit dieses Verfahrens wird anhand einer prototypischen Implementierung beurteilt. Dabei wird eine clientseitige Anwendung aus dem Bereich des eHealths realisiert, die eine kollaborative Bearbeitung von Patientendaten ermöglichen soll. Darüber hinaus wird gemäß der Architektur des Synchronisationsprotokolls auch ein allgemeiner Synchronisationsserver umgesetzt. Nachfolgend wird die gesamte Implementierung und somit auch das Synchronisationsverfahren sowohl anwendungsbezogen als auch technisch evaluiert und den vorhergehenden Ansätzen gegenübergestellt.

Ausblick

Da in dieser Arbeit lediglich einzelne Datensätze aus einer Key-Value-Tabelle über die Gerätegrenze hinaus synchronisiert werden, könnten aufbauende Arbeiten die Synchronisation sämtlicher Datenbanken einschließlich der darin vorkommenden Beziehungen unter Verwendung derselben Synchronisationsstrategie unterstützen. Hierdurch könnten auch komplexere Anwendungen umgesetzt werden, die auf gemeinsam genutzten Daten operieren.

Da bislang Datenkonflikte zur Auflösung vollständig an den Benutzer übergeben werden, könnten auch neue Vorgehensweisen entwickelt werden, die eine autonome Lösungsfindung für spezielle Anwendungsfelder ermöglichen.
Außerdem kann das vorgestellte Synchronisationsprotokoll insofern ergänzt werden, dass Datenbanken mit unterschiedlichen Datenmodellen, wie beispielsweise Graphdatenbanken, auch unterstützt werden.

Literaturverzeichnis

[AHF+09] D. M. Aanensen, D. M. Huntley, E. J. Feil, B. G. Spratt et al. „EpiCollect: linking smartphones to web applications for epidemiology, ecology and community data collection“. In: *PloS one* 4.9 (2009), e6968 (zitiert auf S. 16).

[AST02] S. Agarwal, D. Starobinski, A. Trachtenberg. „On the scalability of data synchronization protocols for PDAs and mobile devices“. In: *IEEE network* 16.4 (2002), S. 22–28 (zitiert auf S. 37).

[ATA13] T. A. Alhaj, M. M. Taha, F. M. Alim. „Synchronization wireless algorithm based on message digest (SWAMD) for mobile device database“. In: *Computing, Electrical and Electronics Engineering (ICCEEE), 2013 International Conference on*. IEEE. 2013, S. 259–262 (zitiert auf S. 24, 27).

[CCP+10] M.-Y. Choi, E.-A. Cho, D.-H. Park, C.-J. Moon, D.-K. Baik. „A database synchronization algorithm for mobile devices“. In: *IEEE Transactions on Consumer Electronics* 56.2 (2010) (zitiert auf S. 16, 17, 23–25, 27, 74).

[Cro10] S. A. Cromar. „Smartphones in the US: market analysis“. In: (2010) (zitiert auf S. 11).

[Eng02] T. R. Eng. „eHealth research and evaluation: challenges and opportunities“. In: *Journal of health communication* 7.4 (2002), S. 267–272 (zitiert auf S. 19).

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. *Hypertext transfer protocol–HTTP/1.1*. Techn. Ber. 1999 (zitiert auf S. 40).

[GR92] J. Gray, A. Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992 (zitiert auf S. 12).

[HK12] G. de Heer, S. Kluge. „Kommunikation in der Intensivmedizin“. In: (2012) (zitiert auf S. 20).

[Höp01] H. Höpfner. „Replikation in mobilen Datenbanken.“ In: *BTW Studierenden-Programm*. 2001, S. 7–9 (zitiert auf S. 31, 32).

[JB12] M. Jaekel, K. Bronnert. *Die digitale Evolution moderner Großstädte: Apps-basierte innovative Geschäftsmodelle für neue Urbanität*. Springer-Verlag, 2012 (zitiert auf S. 11).

[Jep97] B. Jepson. *Java database programming*. Wiley, 1997 (zitiert auf S. 12).

[JSY09] L. Junyan, X. Shiguo, L. Yijie. „Application research of embedded database SQLite“. In: *Information Technology and Applications, 2009. IFITA'09. International Forum on*. Bd. 2. IEEE. 2009, S. 539–543 (zitiert auf S. 11).

[ML02] F. Matthes, V. Lehel. „Dokument-und Kontaktsynchronisation mit mobilen Datenbanken: Anforderungen und Lösungsansätze aus Sicht von Unternehmensportalen (DRAFT).“ In: *Mobile Datenbanken und Informationssysteme*. 2002, S. 3–9 (zitiert auf S. 16).

[Owe03] M. Owens. „Embedding an SQL database with SQLite“. In: *Linux Journal* 2003.110 (2003), S. 2 (zitiert auf S. 15).

[PN04] S. H. Phatak, B. Nath. „Transaction-centric reconciliation in disconnected client–server databases“. In: *Mobile Networks and Applications* 9.5 (2004), S. 459–471 (zitiert auf S. 32).

[PR85] J. Postel, J. Reynolds. „File transfer protocol“. In: (1985) (zitiert auf S. 40).

[PS11] C. Pettey, H. Stevens. „Gartner says sales of mobile devices grew 5.6 percent in third quarter of 2011; smartphone sales increased 42 percent“. In: *Retrieved August* 19 (2011), S. 2012 (zitiert auf S. 15).

[Rah94] E. Rahm. *Mehrrechner-Datenbanksysteme: Grundlagen der verteilten und parallelen Datenbankverarbeitung*. Erhard Rahm, 1994 (zitiert auf S. 73).

[RBB17] A. Rottmann, B. Bumgarner, L. Boszormenyi. *SQLITE3(1) Linux User's Manual*. 2017 (zitiert auf S. 15).

[SM13] C. Stach, B. Mitschang. „Privacy management for mobile platforms–a review of concepts and approaches“. In: *Mobile Data Management (MDM), 2013 IEEE 14th International Conference on*. Bd. 1. IEEE. 2013, S. 305–313 (zitiert auf S. 34).

[SM14] C. Stach, B. Mitschang. „Design and implementation of the privacy management platform“. In: *Mobile Data Management (MDM), 2014 IEEE 15th International Conference on*. Bd. 1. IEEE. 2014, S. 69–72 (zitiert auf S. 34).

[SM15] C. Stach, B. Mitschang. „Der Secure Data Container (SDC) - Sicheres Datenmanagement für mobile Anwendungen“. In: *Datenbank-Spektrum* 15.2 (2015), S. 109–118. ISSN: 1618-2162. DOI: 10.1007/s13222-015-0189-y. URL: http://dx.doi.org/10.1007/s13222-015-0189-y (zitiert auf S. 34, 75).

[SM16] C. Stach, B. Mitschang. „The Secure Data Container: An Approach to Harmonize Data Sharing with Information Security“. In: *Proceedings of the 2016 IEEE 17th International Conference on Mobile Data Management*. MDM '16. IEEE, Juni 2016, S. 292–297. DOI: 10.1109/MDM.2016.50 (zitiert auf S. 34).

[SM18] C. Stach, B. Mitschang. „CURATOR—A Secure Shared Object Store: Design, Implementation, and Evaluation of a Manageable, Secure, and Performant Data Exchange Mechanism for Smart Devices“. In: *Proceedings of the 33rd ACM/SIGAPP Symposium On Applied Computing*. DTTA '18. ACM, 2018, S. 533–540. DOI: 10.1145/3167132.3167190 (zitiert auf S. 34).

[SMC+14] D. Sethia, S. Mehta, A. Chowdhary, K. Bhatt, S. Bhatnagar. „MRDMS-mobile replicated database management synchronization“. In: *Signal Processing and Integrated Networks (SPIN), 2014 International Conference on*. IEEE. 2014, S. 624–631 (zitiert auf S. 28, 30, 31, 75).

[SSPE04] C. Spyrou, G. Samaras, E. Pitoura, P. Evripidou. „Mobile agents for wireless computing: the convergence of wireless computational models with mobile-agent technologies“. In: *Mobile Networks and Applications* 9.5 (2004), S. 517–528 (zitiert auf S. 12, 77).

[TDP+94] M. Theimer, A. Demers, K. Petersen, M. Spreitzer, D. Terry, B. Welch. „Dealing with tentative data values in disconnected work groups“. In: *Mobile Computing Systems and Applications, 1994. Proceedings., Workshop on*. IEEE. 1994, S. 192–195 (zitiert auf S. 18).

[Tur14] S. Turner. „Transport layer security“. In: *IEEE Internet Computing* 18.6 (2014), S. 60–63 (zitiert auf S. 34).

[Ven14] C. L. Ventola. „Mobile devices and apps for health care professionals: uses and benefits“. In: *Pharmacy and Therapeutics* 39.5 (2014), S. 356 (zitiert auf S. 11).

[XDC+14] B. Xu, L. Da Xu, H. Cai, C. Xie, J. Hu, F. Bu. „Ubiquitous data accessing method in IoT-based information system for emergency medical services“. In: *IEEE Transactions on Industrial Informatics* 10.2 (2014), S. 1578–1586 (zitiert auf S. 11).

Alle URLs wurden zuletzt am 04. 05. 2018 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift