Evaluation von Synchronisations- und Konfliktlösungsverfahren im Web-Umfeld

Martin Eigenmann

2. Juni 2015

1

Abstract

2

Danksagungen

Zunächst möchte ich an all diejenigen meinen Dank richten, die mich während der Durchführung der Thesis unterstützt und motiviert haben. Auch ist allen, die während des Studiums mehr Geduld und Verständnis für mich aufbrachten ein ganz spezieller Dank geschuldet.

Ganz besonders möchte ich mich bei meinem Betreuer Philip Stanik bedanken, der immer vollstes Vertrauen in meine Fähigkeiten besass, mich durch kritisches Hinterfragen und Anregungen zum richtigen Zeitpunkt bestens unterstützt hat.

Auch meinem Arbeitgeber ist an diesem Punkt ein grosses Dankeschön für die gute und freundschaftliche Unterstützung geschuldet. Ohne die Flexibilität des Vorgesetzten und der Mitarbeiter wäre diese Arbeit nicht durchführbar gewesen.

Nicht zuletzt gebührt auch meinen Eltern Dank, ohne die ich das Studium nicht durchgestanden hätte.

3

Inhaltsverzeichnis

1 Abstract **i**

2 Danksagungen **ii**

3 Inhaltsverzeichnis **iii**

4 Personalienblatt **vi**

5 Bestätigung **vii**

I Präambel **1**

6 Einleitung **2**
6.1 Motivation und Fragestellung 2
6.2 Aufgabenstellung 3
6.3 Abgrenzung der Arbeit 3

7 Aufbau der Arbeit **4**

II Grundlagen **5**

8 Recherche **6**
8.1 Fachbegriffe 6
8.2 Erläuterung der Grundlagen 6
8.3 Replikationsverfahren 9
8.4 Synchronisationsverfahren 10

9 Analyse **12**
9.1 Synchronisationsproblem 12
9.2 Datenanalyse 13
9.3 Datenanalyse der Synchronisationsprobleme 15

9.4 Überprüfung der Klassifikation 15

III Konzept **17**

10 Konzeptansätze **18**
10.1 Synchronisation 18
10.2 Datenhaltung 18
10.3 Konfliktvermeidung 20
10.4 Konfliktauflösun 21

11 Konzept Untersuchung **24**
11.1 Synchronsation 24
11.2 Datenhaltung 24
11.3 Konfliktvermeidung 26
11.4 Konfliktauflösung 27
11.5 Zusammenfassung 29

12 Leitfaden **30**
12.1 Busines-Logic muss Sync/Konflikte vorsehen 30
12.2 Verwenden von persönlichen Daten 30
12.3 verwenden von Insert statt Update 30
12.4 verwenden von altmodischem Lock? 30
12.5 Deaktivieren von Features wenn offline 30

13 Design des Prototypen **31**
13.1 Design-Ansätze 31
13.2 Entscheid 31
13.3 Design 32
13.4 Beispielapplikation 35

IV Implementation **36**

14 Prototyp **37**
14.1 Umsetzung 37
14.2 Technologie Stack 37
14.3 Entwicklungsumgebung 38
14.4 Entwicklung 38

15 Testing **40**

V Ausklang **41**

16 Review **42**
16.1 Validation 42
16.2 Ausblick 42

17 Fazit und Schlusswort **43**

A Anhang **44**

A.1 Glossar 44
A.2 Aufgabenstellung 44
A.3 Detailanalyse der Aufgabenstellung 46

B Projektmanagement **48**
B.1 Projektplanung 48
B.2 Termine 48
B.3 Dokumentation 49
B.4 Versionsverwaltung 49

C Anforderungsanalyse **50**
C.1 Vorgehensweise 50
C.2 Risiken 54

D Verzeichnisse **55**
D.1 Quellenverzeichnis 55
D.2 Tabellenverzeichnis 56
D.3 Abbildungsverzeichnis 56

4

Personalienblatt

Name, Vorname	Eigenmann, Martin
Adresse	Harfenbergstrasse 5
Wohnort	9000 St.Gallen
Geboren	4. Juli 1990
Heimatord	Waldkirch

5
Bestätigung

Hiermit versichere ich, die vorliegende Bachelorthesis eigenständig und ausschliesslich unter Verwendung der angegebenen Hilfsmittel angefertigt zu haben.
Alle öffentlichen Quellen sind als solche kenntlich gemacht. Die vorliegende Arbeit ist in dieser oder anderer Form zuvor nicht als Semesterarbeit zur Begutachtung vorgelegt worden.

St.Gallen 1.05.2015

Martin Eigenmann

Teil I.

Einleitung und Abgrenzung

6
Einleitung

6.1. Motivation und Fragestellung

Der Zugriff auf Services und Medien mittels mobiler Geräte steigt beständig an. So ist im Mai 2014, 60% der Zeit, die online verbracht wird, über Handy und Tablet zugegriffen worden - davon 51% mittels mobiler Applikationen. [6]

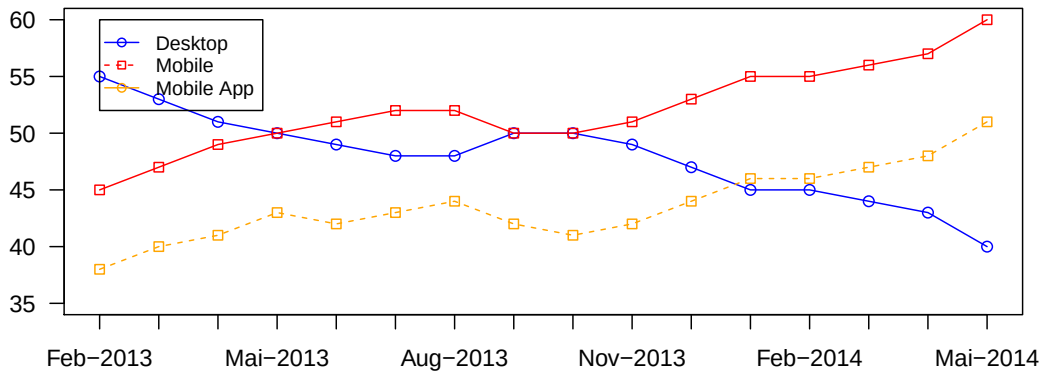


Abbildung 6.1.: Verteilung der online verbrachten Zeit nach Platform (Grafik erstellt gemäss der Daten von [6])

Angesichts der grossen Verbreitung und Nutzung von Services und Medien im Internet, wird eine unterbruchsfreie Benutzbarkeit, auch ohne Internetverbindung, immer selbstverständlicher und somit auch immer wichtiger.

> „It's clear that the mobile industry has finally given up on the fantasy that an Internet connection is available to all users at all times. Reality has set in. And in

> the past month, we've seen a new wave of products and services that help us go offline and still function.“ Elgan [3]

Es stellt sich nun die Frage, wie Informationen, über Verbindungsunterbrüche hinweg, integer gehalten werden können. Und wie Daten im mobilen Umfeld synchronisiert, aktualisiert und verwaltet werden könne, so dass, für den Endbenutzer schlussendlich kein Unterschied zwischen Online- und Offline-Betrieb mehr wahrnehmbar ist.

Während der Bearbeitung meiner Semesterarbeit hatte ich mich bereits mit der Synchronisation von Daten zwischen Backend und Frontend beschäftigt, wobei aber der Fokus klar auf der Logik des Backends lag. Die erarbeitet Lösung setzte eine ständige Verbindung zwischen Client und Server voraus, was bei der praktischen Umsetzung zu erheblichen Einschränkungen für die Benutzer führte.

Mein starkes persönliches Interesse zur Analyse und Verbesserung dieser Prozesse treibt mich an, diese Arbeit durch zu führen.

6.2. Aufgabenstellung

Die von der Studiengangsleitung für Informatik freigegebene Aufgabenstellung sowie eine Detailanalyse derer ist im Appendix unter „Aufgabenstellung“ aufgeführt.

6.3. Abgrenzung der Arbeit

Grosse Anbieter von Web-Software wie Google und Facebook arbeiten intensiv an spezifischen Lösungen für ihre Produkte. Zwar werden in Talks Techniken und Lösungsansätze erläutert (Facebook stellt Flux und Message-Driven Architecture vor[1]), wissenschaftliche Arbeiten darüber, sind jedoch nicht publiziert.

Ich möchte in dieser Arbeit einen allgemeinen Ansatz erarbeiten und die Grenzen ausloten, um so zu zeigen, wo die Limitationen der Synchronisation liegen.

Die Arbeit eröffnet mit ihrer Fragestellung ein riesiges Gebiet und wirft neue Fragestellungen auf. Die ursprüngliche Fragestellung wird vertieft behandelt, ohne auf Neue in gleichem Masse einzugehen.
Die zu synchronisierenden „Real World“ Fälle sind sehr unterschiedlich und nicht generalisierbar. Es sind zwei exemplarische Anwendungsfälle erarbeitet, auf welchen die Untersuchungen durchgeführt werden.

Zusätzlich wird die Arbeit durch folgende Punkte klar abgegrenzt:

- Die Informationsbeschaffung findet ausschliesslich in öffentlich zugänglichen Bereichen statt. (Internet/Bibliothek)
- Grundsätzliches Wissen zu Prozessen und Frameworks wird vorausgesetzt und nur an Schlüsselstellen näher erklärt.

[1] https://www.youtube.com/watch?v=KtmjkCuV-EU

7
Aufbau der Arbeit

*Teil i - **Präambel** - Einleitung und Abgrenzung*

*Teil ii - **Grundlagen** - Technische Grundlagen und Architekturen*
Das Kapitel Recherche setzt sich mit den Grundlagen, Fachbegriffen und bekannten Verfahren zur Synchronisation und Replikation auseinander.
Im Kapitel Analyse werden die Grundlagen geschaffen um darauf Konzepte zu erstellen. Es wird eine Analyse der zu synchronisierenden Daten erstellt und überprüft ob eine sinnvolle Klassifikation durchführbar ist.

*Teil iii - **Konzept** - Konzeption und Konzeptüberprüfung*
In diesem Teil der Thesis...
Ich überprüfe die erarbeiten Konzepte auch auf deren Wirksamkeit...

*Teil iv - **Implementation** - Implementierung und Testing*
Dieser Teil beschäftigt sich mit der Entwicklung des Prototypen. Obwohl Testing und Implementation immer zusammen umgesetzt werden, ist in dieser Arbeit eine Aufteilung in die beiden Kapitel Prototyp und Testing gewählt, um dem Leser...

*Teil v - **Ausklang** - Abschluss und Ausblick*
In diesem Teil wird evaluiert, ob die erarbeiten und implementierten Konzepte der Aufgabenstellung genügen...

Teil II.

Technische Grundlagen und Architekturen

8
Recherche

8.1. Fachbegriffe

Eine Aufführung und dazugehörende Ernährung der für das Verständnis der Arbeit notwendigen Fachbegriffe befindet sich im Anhang unter dem Kapitel „Glossar“.

8.2. Erläuterung der Grundlagen

In diesem Kapitel werden Funktionsweisen und Grundlage ausgeführt, die als für die Bearbeitung dieser Bachlorthesis herangezogen wurden.

8.2.1. Datenbanken

Eine Datenbank [1] ist ein System zur Verwaltung und Speicherung von strukturierten Daten. Erst durch den Kontext des Datenbankschemas wird aus den Daten Informationen, die zur weiteren Verarbeitung genutzt werden können. Ein Datenbanksystem umfasst die beiden Komponenten Datenbankmanagementsystem (DBMS) sowie die zu veraltenden Daten selbst.
Ein DBMS muss die vier Aufgaben [2] erfüllen.

- Atomarität
- Konsistenzerhaltung
- Isolation
- Dauerhaftigkeit

Neben den vielen neu auf den Markt erschienen Technologien wie Document Store oder Key-Value Store ist das Relationale Datenbankmodell immer noch am weitesten verbreitet. [2].

[1] In der Literatur oft auch als **D**aten**B**ank**S**ystemen (DBS) oder Informationssystem bezeichnet. [7, pp. 3-4]
[2] Bekannt als ACID-Prinzip [7, pp.105] umfasst es **A**tomicity, **C**onsistency, **I**solation und **D**urability.

8.2.2. Monolithische Systeme

Als Monolithisch wird ein logisches System bezeichnet, wenn es in sich geschlossen, ohne Abhängigkeiten zu anderen Systemen operiert. Alle zur Erfüllung der Aufgaben benötigten Ressourcen sind im System selbst enthalten. Es müssen also keine Ressourcen anderer Systeme alloziert werden und somit ist auch keine Kommunikation oder Vernetzung notwendig.
Das System selbst muss jedoch nicht notwendigerweise aus nur einem Rechenknoten bestehen, sondern darf auch als Cluster implementiert sein.

8.2.3. Verteilte Systeme

Man kann zwischen physisch und logisch verteilten Systemen unterscheiden. Weiter kann das System auf verschiedenen Abstraktionsstufen betrachtet werden. So sind je nach Betrachtungsvektor unterschiedliche Aspekte relevant und interessant. [11]

physisch verteilte Systeme

Rechnernetze und Cluster-Systeme werden typischerweise als physisch verteiltes System betrachtet. Die Kommunikation zwischen den einzelnen Rechenknoten erfolgt nachrichtenorientiert und ist somit asynchron ausgelegt. Jeder Rechenknoten verfügt über exklusive Speicherressourcen und einen eigenen Zeitgeber.
Durch die Implementation eines Systems über mehrere unabhängige physische Rechenknoten kann eine erhöhte Ausfallsicherheit und/oder ein Performance-Gewinn erreicht werden.

logisch verteilte Systeme

Falls innerhalb eines Rechenknoten echte Nebenläufigkeit[3] oder Modularität[4] erreicht wird, kann von einem logisch verteilten System gesprochen werden. Einzelne Rechenschritte und Aufgaben werden unabhängig voneinander auf der selben Hardware ausgeführt. Dies ermöglicht den flexiblen Austausch[5] einzelner Module.

8.2.4. Verteilte Algorithmen

Verteilte Algorithmen sind Prozesse welche miteinander über Nachrichten (synchron oder asynchron) kommunizieren und so idealerweise ohne Zentrale Kontrolle eine Kooperation erreichen. [10]
Performance-Gewinn, bessere Skalierbarkeit und eine bessere Unterstützung von verschiedenen Hardware-Architekturen kann durch den Einsatz von verteilten Algorithmen erreicht werden.

[3] Von echter Nebenläufigkeit wird gesprochen, wenn verschiedene Prozesse zur selben Zeit ausgeführt werden. (Multiprozessor)

[4] Modularität beschreibt die Unabhängigkeit und Austauschbarkeit einzelner (Software-) Komponenten. (Auch Lose Kopplung gennant)

[5] Austauschbarkeit einzelner Programmteile wird durch die Einhaltung der Grundsätze von modularer Programmierung erreicht.

8.2.5. Replikation

Replikation vervielfacht ein sich möglicherweise mutierendes Objekt (Datei, Dateisystem, Datenbank usw.), um hohe Verfügbarkeit, hohe Performance, hohe Integrität oder eine beliebige Kombination davon zu erreichen. [1, p. 19]

Synchrone Replikation

Eine synchrone Replikation stellt sicher, dass zu jeder Zeit der gesamte Objektbestand auf allen Replikationsteilnehmern identisch ist.

Wird ein Objekt eines Replikationsteilnehmers mutiert, wird zum erfolgreichen Abschluss dieser Transaktion, von allen anderen Replikationsteilnehmern verlangt, dass sie diese Operation ebenfalls erfolgreich abschliessen.
Üblicherweise wird dies über ein Primary-Backup Verfahren realisiert. Andere Verfahren wie der 2-Phase-Commit und 3-Phase-Commit ermöglichen darüber hinaus auch das synchrone Editieren von Objekten auf allen Replikationsteilnehmern. [1, p. 23ff, 134ff]

Asynchrone Replikation

Eine asynchrone Replikation, stellt periodisch sicher, dass der gesamte Objektbestand auf allen Replikationsteilnehmern identisch ist. Mutationen können nur auf dem Master-Knoten durchgeführt werden. Einer oder mehrere Backup-Knoten übernehmen dann periodisch die Mutationen. Entgegen der synchronen Replikation müssen nicht alle Replikationsteilnehmer zu jedem Zeitpunkt verfügbar sein[6].

Merge Replikation

Die merge Replikation erlaubt das mutieren der Objekte auf jedem beliebigen Replikationsteilnehmer.
Mutationen auf einem einzelnen Replikationsteilnehmer werden periodisch allen übrigen Replikationsteilnehmern mitgeteilt. Da ein Objekt zwischenzeitlich[7] auch auf anderen Teilnehmern mutiert worden sein kann, müssen während des Synchronisationsvorgang[8] eventuell auftretenden Konflikte aufgelöst werden.

8.2.6. Block-Chain

Die Block-Chain ist eine verteilte Datenbank die ohne Zentrale Autorität auskommt. Jede Transaktion wird kryptographisch gesichert der Kette von Transaktionen hinzugefügt. So ist das entfernen oder ändern vorhergehender Einträge nicht mehr möglich[9]. Jeder Teilnehmer darf

[6] So kann der Backup-Knoten nur Nachts über verfügbar sein, damit der dazwischen liegende Kommunikationsweg Tags über nicht belastet wird.

[7] Zwischen der lokalen Mutation und der Publikation dieser an die übrigen Replikationsteilnehmer, liegt eine beliebige Latenz.

[8] Da die Replikation nicht notwendigerweise nur unidirektional, sondern im Falle von einem Multi-Master Setup auch bidirektional durchgeführt werden kann, wird hier von einer Synchronisation gesprochen.

[9] Das ändern vorhergehender Einträge benötigt mehr Rechenzeit, als alle anderen Teilnehmer ab dem Zeitpunkt des Hinzufügens des Eintrages, zusammen aufgewendet haben.

also alle Einträge lesen und neue Einträge hinzufügen. Da Einträge nur hinzugefügt werden und nie ein Eintrag geändert wird, kann eine Block-Chain immer ohne Synchronisationskonflikte repliziert werden. Konflikte können nur in den darüberlegenden logischen Schichten[10] auftreten. [9]

8.3. Replikationsverfahren

8.3.1. MySQL

Das Datenbanksystem MySQL unterstützt asynchrone als auch synchrone Replikation. Beide Betriebsmodi können entweder in der Master-Master[11] oder in der Master-Slave Konfiguration betreiben werden.

Die Master-Slave Replikation unterstützt nur einen einzigen Master und daher werden Mutationen am Datenbestand nur vom Master entgegengenommen und verarbeitet. Ein Slave-Knoten kann aber im Fehlerfall des Masters, sich selbst zum Master befördern.

Der Master-Master Betrieb erlaubt die Mutation des Datenbestand auf allen Replikationsteilnehmern. Dies wird mit dem 2-Phase-Commit (2PC) Protokoll erreicht. Somit sind Konflikte beim Betrieb eines Master-Master Systems ausgeschlossen.

2-Phase-Commit Protokoll

Um eine Transaktion erfolgreich abzuschliessen, müssen alle daran beteiligten Datenbanksysteme bekannt und in einem Zustand sein, in dem sie die Transaktion durchführen (commit) oder nicht (roll back). Die Transaktion muss auf allen Datenbanksystemen als eine einzige Atomare Aktion durchgeführt werden.

Das Protokoll unterscheidet zwischen zwei Phasen[8]:

1. In der ersten Phase werden alle Teilnehmer über den bevorstehenden Commit informiert und zeigen an ob sie die Transaktion erfolgreich durchführen können.
2. In der zweiten Phase wird der Commit durchgeführt, sofern alle Teilnehmer dazu im Stande sind.

8.3.2. MongoDB

MongoDB unterstützt die Konfiguration eines Replica-Sets (Master-Slave) nur im asynchronen Modi. Das Transations-Log des Masters wird auf die Slaves kopiert, welche dann die Transaktionen nachführen. Ein Master-Master Betrieb ist nicht vorhergesehen.

[10] So prüft die Bitcoin-Implementation ob eine Transaktion (Überweisung eines Betrags) bereits schon einmal ausgeführt wurde, und verweigert gegebenenfalls eine erneute Überweisung.

[11] Oft wird Master-Master Replikation auch als Active-Active Replikation referenziert.

8.4. Synchronisationsverfahren

8.4.1. Backbone.js

Das Web-Framework Backbone.js implementiert zur Datenhaltung und Synchronisation einen Key-Value Store. Der Key-Value Store kann entweder ein einzelnes Objekt oder ein gesamtes Set an Objekten[12] von der REST-API lesen sowie ein einzelnes Objekt schreibend an die REST-API senden.

Der REST-Standard[4] verlangt dass für die Übermittlung eines Objekt immer der vollständige Status übermittelt wird. So wird bei lesendem und schreibenden Zugriff immer die gesamte Objekt kopiert.

Backbone.js sieht keinen Mechanismus vor, die auf dem Server stattfindenden Änderungen automatisch auf den Client zu übernehmen. Es muss entweder periodisch die gesamte Collection gelesen oder auf ein Änderungs-Log zugegriffen werden.
Darüber hinaus entscheidet der Server alleine, ob eine konkurrierende Version des Clients übernommen wird. Backbone.js sendet beim Synchronisieren eines Objekts, dessen gesamten Inhalt an den Server und übernimmt anschliessend die von der REST-API zurückgelieferten Version. Die zurückgelieferte Version kann eine auf dem Server bereits zuvor als aktiv gesetzte Version des Objekts sein und somit die gesendeten Aktualisierung gar nicht enthalten[13].

Das Senden einer Aktualisierung an den Server wird mit drei Schritten erledigt.

1. Im ersten Schritt wird ein neues oder mutiertes Objekt an den Server übermittelt.
2. Der Server entscheidet im zweiten Schritt ob die Änderung komplett oder überhaupt nicht übernommen wird. Wird eine Änderung komplett übernommen wird dem Client dies so bestätigt.
   Wird eine Änderung abgelehnt wird der Client darüber informiert.
3. Im Nachgang wird kann das Objekt vom Client erneut angefordert werden, und so die aktuellste Version zu beziehen.

8.4.2. Meteor.js

Das WEB-Framework Meteor.js implementiert eine Mongo-Light-DB im Client-Teil und synchronisiert diese über das Distributed Data Protokoll mit der auf dem Server liegenden MongoDB in Echtzeit.

Meteor.js setzt auf Optimistic Concurrency. So können Mutationen auf der Client-Datenbank durchgeführt werden und diese erst zeitlich versetzt mit dem Server synchronisiert werden. Je geringer die zeitliche Verzögerung, desto geringer ist die Wahrscheinlichkeit, dass das mutierte Objekt auch bereits auf dem Server geändert wurde. Um eine geringe zeitliche Verzögerung zu erreichen, sind alle Clients permanent mit dem Server mittels Websockets verbunden.

Der Server alleine entscheidet welche Version als aktiv übernommen wird. Somit kann nicht garantiert werden, dass alle vorgenommenen Änderungen auf dem Client erfolgreich an den

[12] Backbone.js verwendet die Begriffe Model für ein einzelnes Objekt, wobei einem Objekt genau ein Key, aber mehrere Values zugeordnet werden können, und Collection für eine Sammlung an Objekten.

[13] Auf dem Server können alle Requests aufgezeichnet werden, um Mutationen nicht zu verlieren. Dies wird vom Standard aber nicht vorausgesetzt und ist ein applikatorisches Problem.

Server übermittelt und übernommen werden. Da Mutationen üblicherweise zeitnah übertragen werden, treten aber nur in seltenen Fällen Konflikte auf.

Die Synchronisation einer Anpassung eines Objekts benötigt drei Schritte.

1. Im ersten Schritt wird ein neues oder die mutierten Attribute eines Objekts an den Server übermittelt.
2. Der Server entscheidet im zweiten Schritt ob die übermittelten Änderungen komplett, teilweise oder nicht angenommen werden.
3. Im dritten Schritt wird dem Client mitgeteilt welche Änderungen angenommen wurden. Der Client aktualisiert sein lokales Objekt entsprechen.

9 Analyse

Bekannte Synchronisationsverfahren betrachten Daten als eine homogene Masse und unterscheide nicht. Konfliktlösungen werden erst auf der Applikationsebene vorgenommen und dann spezifisch auf die Applikationsdaten angewandt.
Es soll begutachtet werden, ob eine generelle Unterteilung der Daten möglich ist, und ob dies einen Nutzen mit sich bringt.

9.1. Synchronisationsproblem

Generell liegt ein Synchronisationskonflikt vor, sobald Daten über einen Kommunikationskanal übertragen werden müssen und für die Zeit der Übertragung kein „Lock“ gesetzt werden kann. Also immer dann, wenn sich Daten während einer Transaktion ändern können.

Um ein Konzept zur Verhinderung und Lösung Synchronisationskonflikten zu erarbeiten, muss zuerst eine Analyse der zu synchronisierenden Daten erstellt werden.
Die Datenanalyse wird mit Bezug auf die zwei im Folgenden aufgeführten Problemstellungen durchgeführt. Beide Problemstellungen sind so gewählt, dass sie zusammen einen möglichst allgemeinen Fall abdecken und so die Überprüfung aussagekräftig bleibt.

9.1.1. Synchronisation von Kontakten

In vielen Anwendungsszenarien müssen Kontaktdaten abgeglichen werden.

Eine Firma X betreibt eine zentrale Kontaktdatenbank an ihrem Hauptsitz.
Verkaufsmitarbeiter müssen jederzeit Kontaktdaten abfragen, neu erfassen und anpassen können, ohne dafür mit dem zentralen Server verbunden zu sein. Gerade in wenig entwickelten oder repressiven Ländern ist eine ständige Verbindung nicht immer gegeben und somit ein Off-Line Modus notwendig.
Die Anpassungen können zu einem späteren Zeitpunkt abgeglichen werden.

Ein Kontakt selbst umfasst die in der Tabelle „Attribute Firmen-Kontakt“ beschriebenen Attribute.

Tabelle 9.1.: Attribute Firmen-Kontakt

Attribut	Beschreibung
Name	Der gesamte Name (Vor-, Nach,- und Mittelname) der (Kontakt-)Person.
Adresse	Die vollständige Postadresse der Firma oder Person.
Email	Alle aktiven und inaktiven Email-Adressen des Kontakts. Jeweils nur eine Email ist die primäre Email-Adresse.
Telefon	Alle aktiven und inaktiven Telefonnummern des Kontakts. Jeweils nur eine Telefonnummer ist die primäre Nummer.
pNotes	Persönliche Bemerkungen zum Kontakt. Nur der Autor einer Notiz, kann diese bearbeiten oder lesen.

9.1.2. Syncrhonisation eines Service Desks

In vielen IT-Organisationen kommt ein Service-Deskt zum Einsatzt. Das Bearbeiten der Support-Fälle und erfassen von Arbeitszeiten muss online, direkt im System erfolgen. Dies kann vor allem fürs Arbeiten ausser Haus, zum Beispiel auf Reisen oder direkt beim Kunden, eine grosse Einschränkung sein.

Ein Support-Fall selbst umfasst die in der Tabelle „Attribute Support-Fall“ beschriebenen Attribute. Es sind nur die für die aufgeführten Szenarien nötigen Attribute erfasst.

Tabelle 9.2.: Attribute Support-Fall

Attribut	Beschreibung
Titel	Titel des Support-Falls.
Beschreibung	Fehler/Problembeschreibung des Tickets.
Anmerkungen	Alle Antworten von Technikern und Kunden. Eine Antwort des Technikers kann als FAQ-Eintrag markiert werden.
Arbeitszeit	Alle erfassten und aufgewendeten Stunden für den Support-Fall.
tArbeitszeit	Das Total der erfassten Arbeitszeit.
pNotes	Persönliche Bemerkungen zum Support-Fall. Nur der Autor einer Notiz, kann diese bearbeiten oder lesen.

9.2. Datenanalyse

Daten können bezüglich ihrer Beschaffenheit, Geltungsbereich und Gültigkeitsdauer unterschieden werden. Im Weiteren wird dies als die Klassifikation beschrieben. Die Datentypisierung hingegen, unterscheidet nach dem äusseren Erscheinungsbild der Daten.

Zur Klassifikation werden nur die in den Daten enthaltenen Informationen herangezogen. Der Datentyp selbst ist dabei unerheblich.

Weiter kann die Klassifikation zwischen Struktur und Art der Daten unterscheiden.

9.2.1. Klassifikation nach Art

Um Daten nach ihrer Art zu Klassifizieren reicht es zu untersuchen wie die Lese- und Schreibrechte sowie deren Gültigkeitsdauer aussehen.

Exklusive Daten können nur von einem Benutzer bearbeitet, aber von diesem oder vielen Benutzern gelesen werden.

Gemeinsame Daten können von vielen Benutzern gleichzeitig gelesen und bearbeitet werden.

Dynamische Daten werden automatisch von System generiert. Benutzer greifen nur lesend darauf zu.

Statische Daten bleiben über einen grossen Zeitraum hinweg unverändert. Viele Benutzer können diese Daten verändern und lesen.

Temporäre Daten werden von System oder Benutzer generiert und sind nur sehr kurz gültig. Nur der Autor der Daten kann diese lesen.

9.2.2. Klassifikation nach Struktur

Bei der Unterscheidung der Daten nach ihrer Struktur, kann zwischen Kontextbezogenen und Kontextunabhängigen Daten differenziert werden. Die Entscheidung welcher Strukturklasse die Daten angehören ist abhängig vom Verständnis der Daten uns liegt somit im Entscheidungsbereich des Datendesigners.

Kontextunabhängige Daten gewinnen selbst durch andere Daten nicht mehr an Informationsgehalt. Gemeint ist damit, dass durch das zusätzliche Betrachten anderer Informationen nicht mehr Wissen bezüglich des einen Attributs entsteht.

Kontextbezogene Daten weisen nur bezüglich eines bestimmten Kontext einen signifikanten Informationsgehalt auf.

Die Adresse eines Kontakts spezifiziert üblicherweise den Ort und das Haus. Zusammen mit dem Namen wird auch die Wohnung eindeutig identifiziert. Die Adresse besitzt also zusammen mit dem Name einen grösseren Informationsgehalt.
Wohingegen die Adresse den Namen nicht weiter spezifiziert.

9.2.3. Datentypisierung

Die Unterscheidung der Daten nach Datentyp differenziert zwischen **numerischen**, **binären**, **logischen** und **textuellen** Daten. Zur Typisierung wird immer die für den Benutzer sichtbare Darstellung verwendet, also jene Darstellung, in welcher die Daten erfasst wurden.

9.3. Datenanalyse der Synchronisationsprobleme

Nachfolgend sind die Attribute der beiden Beispiele „Synchronisation von Kontakten“ sowie „Synchronisation eines Service Desks“ entsprechend der erarbeiteten Klassifikation und Typisierung zugeordnet.

Synchronisation von Kontakten
Der Name eines Kontakts ist der primäre Identifikator eines Kontakts. Ändert sich dieser, ist es sehr wahrscheinlich, dass sich auch andere Attribute ändern.
Die Attribute Adresse, Email und Telefon sind deshalb abhängig vom Namensattribut. Diese Attribute sind also Kontextuell abhängig vom Identifikator.
Das Attribut pNotes hingegen ist völlig unabhängig, da es nur vom Verfasser gelesen und geschrieben werden kann.

Tabelle 9.3.: Klassifikation Attribute Kontakt

Attribut	Struktur	Art	Typ
Name	Unabhängig	gemeinsam	textuell
Adresse	Abhängig (Name)	gemeinsam	textuell
Email	Abhängig (Name)	gemeinsam	textuell
Telefon	Abhängig (Name)	gemeinsam	textuell
pNotes	Unabhängig	exklusiv	textuell

Synchronisation eines Service Desks
Die beiden Attribute Titel und Beschreibung können nur beim Erfassen eines Support-Falls gesetzt werden. Danach bilden sie zusammen den eindeutigen Identifikator. Anmerkungen werden spezifisch für einen Support-Fall erfasst, und sind deshalb nur im Kontext desselben bedeutungsvoll.
Die Totale Arbeitszeit (tArbeitszeit) wird in Abhängigkeit vom Attribut Arbeitszeit vom System errechnet und kann nicht geändert werden.

Tabelle 9.4.: Klassifikation Attribute Kontakt

Attribut	Struktur	Art	Typ
Titel	Unabhängig	statisch	textuell
Beschreibung	Abhängig (Titel)	statisch	textuell
Anmerkungen	Abhängig (Titel)	gemeinsam	textuell
Arbeitszeit	Unabhängig	exklusiv	numerisch
tArbeitszeit	Unabhängig	dynamisch	numerisch
pNotes	Unabhängig	exklusiv	textuell

9.4. Überprüfung der Klassifikation

Aus der Überprüfung kann die Erkenntnis gewonnen werden, dass die Struktur sowie der Typ der Daten allgemeingültig und nachvollziehbar ist.
Es kann klar zwischen der Typisierung exklusiv, gemeinsam und dynamisch unterschieden werden.

Auch sind die beiden Struktur-Klassen kontextbezogen und kontextunabhängig allgemeingültig und ermöglichen eine Abhängigkeitserkennung zwischen den Attributen.

Die weiteren gefundenen Typen statisch und temporär sind wenig sinnvoll, da der Typ „temporär“ immer auch exklusiv und „statisch“ durch gemeinsam ersetzt werden kann.

Teil III.

Konzeption und Konzeptüberprüfung

10
Konzeptansätze

10.1. Synchronisation

Das grundlegende Idee bei der Synchronisation liegt darin, den Zustand der Servers und des Clients, bezüglich der Daten, identisch zu halten.
Der Zustand der Daten können dabei als Status betrachtet werden. So repräsentiert der Zustand der gesamten Datensammlung zu einem bestimmten Zeitpunkt, einen Status. Aber auch der Zustand eines darin enthaltenen Objekts (z.B. eines Kontakts) wird als eigenständiger Status betrachtet.

Der Begriff der Synchronisation wird also im folgenden als Vorgang betrachtet, welcher Mutationen des Status des Clients, auch am Status des Servers durchführt.

Eine Status-Mutation kann dabei auf dem Server nur auf den selben Status angewendet werden, auf welchen sie auch auf dem Client angewendet wurde. Eine Mutation bezieht sich also immer auf einen bereits existierenden Status.

Zur Durchführung einer Syncrhonisation muss sowohl die Mutations-Funktion, sowie der Status auf welchen sie angewendet wird, gekannt sein. Beide Informationen zusammen werden als eine Einheit betrachtet und als **Nachricht** bezeichnet.

10.2. Datenhaltung

Die Datenhaltung beschäftigt sich mit der Frage, wie Daten verwaltet und wann und wie Mutationen darauf angewendet werden. Die beiden erarbeiteten Konzepte Singlestate und Multistate werden im Folgenden genauer erläutert.

10.2.1. Singlestate

Ein Single-State System erlaubt, nach dem Vorbild traditioneller Datenhaltungssystem, zu jedem Zeitpunkt nur einen einzigen gültigen Zustand.

Eingehende Nachrichten N enthalten sowohl die Mutations-funktion als auch eine Referenz auf welchen Status S_x, diese Mutation angewendet werden soll. Resultiert aus der Anwendung einer Nachricht, ein ungültiger Status, wird diese nicht übernommen. Nachrichten, welche nicht übernommen wurden, müssen im Rahmen der Konfliktauflösung separat behandelt werden.

In Abbildung 10.1 sind die nacheinander eingehenden Nachrichten N_1 bis N_4 dargestellt. Nachricht N_2 sowie N_3 referenzieren auf den Status S_2. Die Anwendung der Mutations-Funktion von N_2 auf S_2 resultiert im gültigen Status S_3.
Die Anwenung der später eingegangene Nachricht N_3 auf S_2 führt zum ungültigen Status S_3' und löst damit einen Synchronisationskonflikt aus.

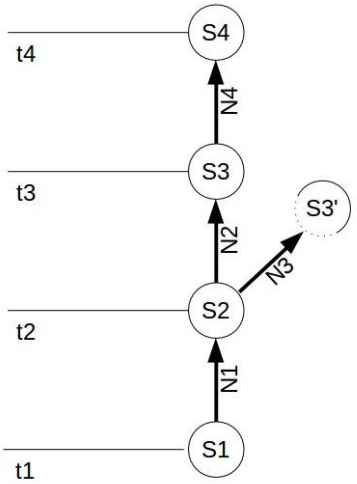


Abbildung 10.1.: Singlestate

Die Konlfiktauflösung hat direkt bei der Anwendung der Mutations-Funktion zu erfolgen und kann nicht korrigiert werden. Die Konfliktauflösung muss also garantieren, dass immer die richtige Entscheidung getroffen wird.

10.2.2. Multistate

Ein Multi-State System erlaubt zu jedem Zeitpunkt beliebig viele gültige Zustände. Zu jedem Zeitpunkt ist jedoch immer nur ein Zustand gültig.

Dieses Verhalten wird dadurch erreicht, dass Zustände rückwirkend eingefügt werden können. Wenn zum Zeitpunkt t_1 und t_2 der gültige Zustand des Systems zum Zeitpunkt t_0 erfragt wird, muss nicht notwendigerweise der identische Zustand zurückgegeben werden.

In der Abbildung 10.2 sind die nacheinander eingehenden Nachrichten N_1 bis N_5 dargestellt. Nachricht N_2 sowie N_4 referenzieren auf den Status S_2. Die Anwendung der Mutations-Funktion von N_2 auf S_2 resultiert im gültigen Status $S_{3.0}$.

Die Anwendung der Nachricht N_3 ergibt folglich den für den Zeitpunkt t_5 gültigen Status $S_{3.1}$.

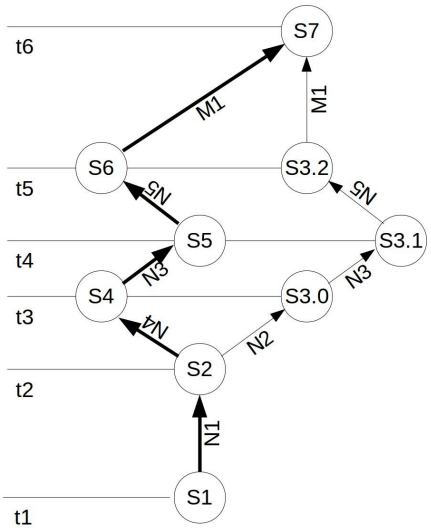


Abbildung 10.2.: Multistate

Der Zeitpunkt an welchem die Nachricht N_4 verarbeitet ist, wird mit hier t_A bezeichnet. Die Anwendung der Nachricht N_4 resultiert im neuen, für den Zeitpunkt t_3, gültigen Status S_4. Vor dem Zeitpunkt t_A ist für den Zeitpunkt t_3 der Status $S_{3.0}$ gültig. Danach ist der Status S_4 gültig.

Die Mutations-Funktion der Nachricht N_3 ist in keinem Konflikt mit der Nachricht N_4 oder N_2 und wird folglich auf den Status S_4 und $S_{3.0}$ angewendet. N_5 steht ebenfalls weder im Konflikt mit N_2 noch mit N_4 und kann desshalb auf S_5 und $S_{3.1}$ angewendet werden.

Die beiden zum Zeitpunkt t_5 gültigen Stati beinhalten das Maximum an Information. Der Status S_6 beinhaltet alle Informationen ausser der Nachricht N_2 und Status $S_{3.2}$ alle Informationen ausser der Nachricht N_4.

Zu einem späteren Zeitpunkt t_6 wird eine Konfliktauflösung M_1 durchgeführt und somit der Konflikt aufgelöst.

Welcher der Zweige bei einer Vergabelung als gültig zu definiert ist, wird über eine Vergabelungs-Funktion beurteilt. Diese gehört zur Konfliktauflösung und ist abhängig von der Datenbeschaffenheit und Struktur der Daten (Kapitel Datenanalyse).

Die Konfliktauflösung kann direkt bei der Anwendung der Mutations-Funktion, oder zu einem beliebigen späteren Zeitpunkt durchgeführt werden. Die Konfliktauflösung darf sehr greedy entscheiden, da Fehlentscheide korrigiert werden können.

10.3. Konfliktvermeidung

Das Konzept der Konfliktvermeidung verhindert das Auftreten von möglichen Konflikten durch die Definition von Einschränkungen im Funktionsumfang der Datenbanktransaktionen. So sind

Objekt aktualisierende Operationen nicht möglich und werden stattdessen, Client seitig, über hinzufügende Operationen ersetzt.

10.3.1. Update Transformation

Damit Mutationen für eines oder mehrere Attribute gleichzeitig konfliktfrei synchronisiert werden können, wird für jedes einzelne Attribut eine Mutations-Funktion erstellt. Die einzelnen Funktionen können in einer Nachricht zusammengefasst werden.

Wie die Abbildung 10.3 zeigt, muss nicht das gesamte Objekt aktualisiert werden und es wird so ermöglicht die Konfliktauflösung granularer durch zu führen.

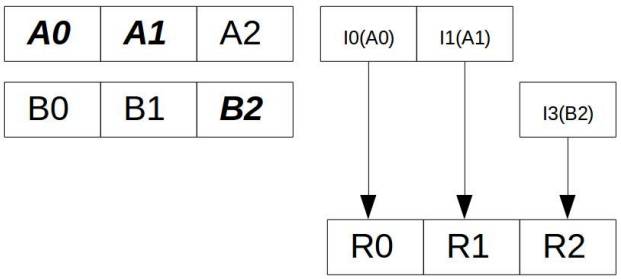


Abbildung 10.3.: Update Transformation

10.3.2. Wiederholbare Transaktion

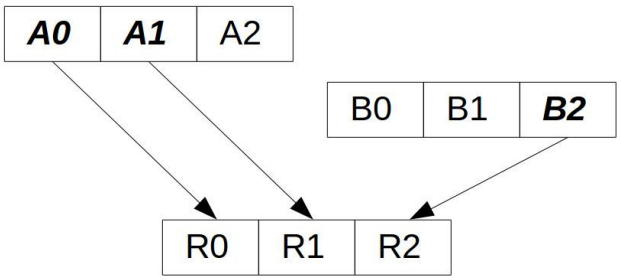
Leseoperationen auf Stati des Clients, welche noch nicht mit dem Server synchronisiert sind, liefern möglicherweise falsche Resultate.
Alle Schreiboperationen, welche Resultate der Leseoperationen mit falschem Resultat, verwenden, dürfen ebenfalls nicht synchronisiert werden, oder müssen mit der korrekten Datenbasis erneut durchgeführt werden.
Dies führt zur Vermeidung von logischen Synchronisationskonflikten.

10.4. Konfliktauflösun

Das Konzept der Konfliktauflösung beschäftigt sich mit der Auflösung von Konflikten, die im Rahmen der Synchronisation aufgetreten sind.
Da die Beschaffenheit und Struktur der Daten, bei dieser Problemstellung eine entscheidende Rolle einnehmen, ist im folgendn für jede Klassifikations-Gruppe ein geeigneter Konfliktauflösungs-Algorithmus aufgeführt.

10.4.1. Zusammenführung

Die Attribute eines Objekts werden einzeln behandelt und auftretende Konflikte separat aufgelöst. So wird nur überprüft ob sich das entsprechende Attribut oder dessen Kontext, falls vorhanden, zwischen dem referenzierten und dem aktuellen Status verändert hat. Falls dies nicht der Fall ist, kann die Mutation konfliktfrei angewendet werden.



10.4.2. geschätze Zusammenführung

Die geschätzte Zusammenführung verwendet einen Algorithmus, welcher das beste Resultat schätzt. Dabei werden alle auf den Status angewendeten Nachrichten analysiert, und das vermutlich beste Resultat verwendet.
Diese Schätzung kann über zum Beispiel mit neuronalen Netzen umgesetzt werden

10.4.3. manuelle Zusammenführung

Wenn keine Auflösung des Konflikts möglich ist, muss dieser manuell aufgelöst werden. Dabei muss das System angehalten werden, oder die Nachricht zurückgehalten werden.

10.4.4. Vergabelungs-Funktion

Bei der Entscheidung welcher Teilbaum aktiv wird können unterschiedliche Vorgehensweisen angewendet werden. Die verwendete Lösung muss auf die Datenbeschaffenheit und Struktur angepasst werden.
Nachfolgende sind fünf Ansätze ausgeführt.

Wichtigste Information

Die Attribute eines Objekts können in aufsteigenden Wichtigkeits-Klassen gruppiert werden. Die Wichtigkeit einer Nachricht entspricht der höchsten Wichtigkeits-Klasse die mutiert wird. Die Nachricht mit der grössten Wichtigkeit markiert den aktiven Teilbaum.

Maximale Information

Den Attributen eines Objekts werden nummerische Informationsgehalts-Indikator zugewiesen. Der Informationsgehalt einer Nachricht entspricht der Summe aller Informationsgehalts-Indikatoren der mutierten Attribute.
Die Nachricht mit dem höheren Informationsgehalt markiert den aktiven Teilbaum.

Geringste Kindsbäume

Für jeden Teilbaum werden die der darin vorkommenden Vergabelungen gezählt. Die Nachricht, die den Teilbaum mit den wenigsten darin vorkommenden Vergabelungen markiert den aktiven Teilbaum.

Online-First

Zusätzlich zur Mutations-Funktion und der Status-Referenz wird einer Nachricht die Information mitgegeben, ob der Client diese online sendet.
Die Nachrichten, welche online gesendet wurden, werden immer den offline synchronisierten Nachrichten vorgezogen.

Timeboxing

Periodisch wird ein Status manuell validiert und als „Grenze“ gesetzt. Nachrichten, welche auf ältere Stati, oder auf Stati in anderen Zweigen referenzieren, markieren nie einen aktiven Teilbaum.

11
Konzept Untersuchung

11.1. Synchronsation

Delta Orientiert

Update Orientiert

11.2. Datenhaltung

Eine sehr elegante Form der Datenhaltung ist die Messagequeue. Der aktuell gültige Staus ist durch die Anwendung aller in der Messagequeue enthaltenen Nachrichten auf den initialen Status erreichbar.
Der wahlfreie Zugriff auf jeden beliebigen Staus zeichnet dieses einfache Design aus. Gerade wegen diesem wahlfreien Zugriff auf beliebige Stati ist diese Form ideal für die Verwendung im Rahmen dieser Thesis geeignet.

Die beiden im Kapitel [Konzept] untersuchten Datenhaltungskonzepte sind nachfolgend genauer untersucht. Gezeigt wird wie ansatzweise eine Implementation aussehen könnte, um Probleme und Vorteile besser erkennen zu können. Konflikt-verhinderung und -auflösung wird in den darauf folgenden Kapiteln genauer untersucht.

11.2.1. Singlestate

Beim Eingehen einer neuen Nachricht wird die Funktion addMessage() aufgerufen.
Die Funktion **State**(t) gibt den Staus zum Zeitpunkt t zurück. Falls kein Zeitpunkt angegeben ist, wird der neueste zurückgegeben.
Die MessageQueue wird durch das Stausobjekt verwaltet.

```
getState(t): ->
    return @State(t)

addMessage(Message): ->
    if Message.refState is @State
        @State().apply Message.Mutation
    else
        if not @State().conflictsWith Message
        or @State().canResolvConfict Message
            @State().apply Message
        else
            break
```

Die Funktionen canResolvConfict sowie resolveConflict greifen auf den referenzierten Status der Nachricht zu.

Performance

Der Zugriff auf einen beliebigen Status ist von der Laufzeitkomplexität $O(n)$ (mit $n =$ Grösse der MessageQueue).

Verbesserung

Da jede schreibende Operation zuerst den referenzierten Staus auslesen muss, und dies sehr Rechenintensiv ist, wird jeder errechneter Zustand zwischengespeichert. So existiert für jede Nachricht bereits ein zwischengespeicherter Status und muss daher nicht für jede Operation erneut generiert werden.

Pro/Contra

Das Konzept des Singlestate ist sehr konservativ und ist in ähnlicher Form weit verbreitet. MongoDB und MySQL bieten beide das Konzept eines einzigen gültigen und unveränderbaren Status. Auch eine Versionierung und somit ein wahlfreier Zugriff auf alle Stati ist implementierbar.

Das grösste Manko liegt jedoch im Umstand, einen Konflikt direkt beim Auftreten auflösen zu müssen. Konflikte die nicht aufgelöst werden können blockieren den gesamten Vorgang oder müssen abgebrochen werden.

11.2.2. Multistate

Beim Eingehen einer neuen Nachricht wird ebenfalls die Funktion addMessage() aufgerufen. Die Funktion **StateTree**(t) gitb den Status zum Zeitpunkt t zurück. Neu wird jedoch die MessageQueue separat geführt, da der Statusbaum bei jeder schreibenden Operation neu aufgebaut werden muss.

```
getState(t): ->
    return @StateTree(t)


addMessage(Message): ->

    @MessageQueue.insert Message
    @StateTree() = new Tree

    for Message in @MessageQueue
        @StateTree().apply Message

    for State in @stateTree
        State.tryToResolvConflict
```

Performance

Da bei jeder schreibenden Operation der gesamte Statusbaum neu aufgebaut wird, weist diese Implemantation eine Laufzeitkomplexität von $O(n)$ (mit $n =$ Grösse der MessageQueue) auf.

Verbesserung 1

Falls eine Nachricht auf einen aktuell gültigen Zustand referenziert, muss der Baum nicht erneut aufgebaut werden, da es ausreichend ist, den Baum nur zu erweitern.

Verbesserung 2

Jede schreibende Operation löst die erneute Generierung des gesamten Statusbaums aus. Um diese rechenintensive Operation zu vereinfachen, wird bei jeder Verzweigung der Zustand gespeichert. Eine Schreibende Aktion, muss so nur noch den betroffenen Teilbaum aktualisieren.

Pro/Contra

Der grösste Gewinn beim Multistate Konzept liegt in der zeitlichen Entkoppelung zwischen Synchronisation und Konfliktauflösung.
Die Richtigkeit, also die Qualität der Information, eines Status wird über die Zeit nur grösser. Und genau darin besteht auch das grösste Problem, denn dadurch ist nicht garantiert dass Abfragen wiederholbare Ergebnisse liefern.

11.3. Konfliktvermeidung

11.3.1. Update Transformation

Die einfachste Implementation einer Update-Transformation besteht darin, sowohl das mutierte Objekt, also auch das Ausgangsobjekt zu übertragen. Implizit wird so eine Mutationsfunktion

übermittelt. Es wird der referenzierte Zustand des Objekts sowie die geänderten Attribute des neuen Status übermittelt.

```
composeMessage(reference, current): ->

    for AttrName, Attribut in current
        if Attribut isnt reference[AttrName]
            Message.Mutation[AttrName] = Attribut

    Message.State = reference

    return Message
```

Pro/Contra

Mutationen können konfliktfrei eingespielt werden, da die Operation automatisiert mit dem neueren Status wiederholt werden kann.
Ein sehr grosses Hindernis besteht aber darin, dass viele Benutzereingaben nur mit einer Zuweisung abgebildet werden können und deshalb die ursprüngliche Daten gar nicht in die Mutationsfunktion miteinbezogen werden.

11.3.2. Wiederholbare Transaktion

Eine sehr triviale Implementation besteht darin, sobald eine Nachricht abgelehnt wird, alle nachfolgenden Nachrichten einer Synchronisation auch abzulehnen und den Client neu zu initialisieren.
Ein ähnliches Konzept ist im Gebiet der Datenbanken auch als Transaktion bekannt. Nur wird hier kein Rollback durchgeführt.

Probleme/Lösungen

Da bei einem nicht auflösbaren Konflikt alle Mutationen gelöscht werden, ist garantiert dass keine auf falschen Daten basierten Mutationen synchronisiert werden.
Aber gerade wegen diesem aggressivem Vorgehen, geht unter Umständen viel an Arbeit verloren.

11.4. Konfliktauflösung

11.4.1. Zusammenführung

Die einfachste Implementation besteht darin, nur geänderte Attribute zu übertragen. So werden Konflikte nur behandelt, wenn das entsprechende Attribut mutiert wurde.

```
resolvConflict (valid, reference, current): ->

    NewState = new State

    for AttrName, Attribut in current
        if reference[AttrName] is valid[AttrName]
        or not context
            NewState[AttrName] = Attribut
        else
            break

    return NewState
```

Probleme/Lösungen

Die wesentlich Idee ist, einzelne Attribute als vollwertige Objekte zu behandeln. So können mehr Informationen übernommen werden.

11.4.2. geschätzte Zusammenführung

Um eine normalisierte Zusammenführung um zu setzten, ist zwingend ein wahlfreier Zugriff auf jeden vorherigen Stati notwendig.

```
resolvConflict (valid, reference, average): ->
    NewState = new State
    Distances = new DistanceArray( average )

    for update in reference
        Distances.add update

    bestUpdate = Distances.smallest
    NewState[bestUpdate.AttrName] = bestUpdate.Attribut

    return NewState
```

Probleme/Lösungen

Entstandene Konflikte können aufgelöst werden, ohne dass manuell eingegriffen werden muss. Die Unsicherheit liegt jedoch darin, dass entweder Ausreisser so nicht akzeptiert werden oder für die vorliegenden Daten (z.B. Telefonnummern, Adressen, . . .) gar nicht erst eine Distanzfunktion erstellt werden kann.
Eine zentrale Einschränkung, liegt jedoch darin, dass diese Art der Zusammenführung nur mit Multistate funktioniert.

11.4.3. manuelle Zusammenführung

Eine manuelle Zusammenführung muss in Form eines GUI implementiert werden, womit ein Benutzer diese durchführen kann.

Probleme/Lösungen

Durch die händische Validation der Daten ist sichergestellt, dass der Konflikt richtig aufgelöst wurde.
Gerade bei grossen Datenbeständen gestaltet sich die Organisation einer Validierung sehr aufwändig.

11.5. Zusammenfassung

11.5.1. Datenhaltung

Obwohl die Idee, Konfliktauflösungen zeitlich entkoppelt von Synchronisationsvorgang zu betreiben, sehr verlockend klingt, überwiegen die Nachteile. Keine garantierte Isolation, keine garantierte Atomarität, und keine garantierten Ergebnisse bei wiederholten Abfragen. Alle Eigenschaften die in Datenbanksystemen als wichtig eingestuft werden, sind hier eingeschränkt oder ausgehebelt.

11.5.2. Konfliktauflösung/Konfliktverhinderung

Sowohl die Wiederholbare Transaktion, als auch die Normalisierte Zusammenführung lösen ansonsten nicht auflösbare Konflikte. Da das Konfliktauflösung jedoch nicht notwendigerweise korrekt sein muss und die Implementation sehr aufwändig ist, ist die Einsetzbarkeit nicht gegeben.

Die übrigen Verfahren wie Update Transformation, Zusammenführung sowie die Kontextbezogene Zusammenführung zeigten sich als einsetzbar.

Tabelle 11.1.: Konzept Vergleich Konfliktverhinderung - Konfliktauflösung

Konzept	**Betrieb**	**Implementation**
Update Transformation	einfach	schwierig
Wiederholbare Transaktion	einfach	sehr schwierig
Zusammenführung	einfach	einfach
Kontext b. Zusamm.	einfach	einfach
Normalisierte Zusamm.	einfach	sehr schwierig
Manuelle Zusamm.	sehr schwierig	einfach

12
Leitfaden

sync wird immer gemacht, wenn lokal daten gecached werden

12.1. Busines-Logic muss Sync/Konflikte vorsehen

Konfliktauflösung mit Benutzerinformation

12.2. Verwenden von persönlichen Daten

Daten sind nur von einer Person editierbar

12.3. verwenden von Insert statt Update

12.4. verwenden von altmodischem Lock?

First locked winns - but all can try

12.5. Deaktivieren von Features wenn offline

13
Design des Prototypen

In diesem Kapitel wird ein Prototyp entworfen, der die Erkenntnisse aus dem Kapitel Konzept Untersuchung so umgesetzt dass sie auch den Anforderungen aus dem Kapitel Anforderungsanalyse genügen.

13.1. Design-Ansätze

Zur Lösung der Aufgabestellung wurden zwei Desing-Ansätzte erarbeitet. Diese in den nachfolgenden Kapiteln kurz erläutert.

13.1.1. Nachrichten-basierte Architektur

Der Prototyp verwendet eine Messagequeue zur Datenhaltung. Es wird somit keine konventionelle Datenbank verwendet. Ein wahlfreier Zugriff auf jeden Status ist möglich.

13.1.2. Modellbasierte Architektur

Der Prototyp emuliert das verhalten einer Messagequeue. Ein wahlfreier zugriff auf jeden Status ist nicht möglich, jedoch kann eine konventionelle Datenbank verwendet werden. Es wird sichergestellt dass versendete Nachrichten eine komplette Kopie der referenzierten Status enthalten.

13.2. Entscheid

Die Modellbasierte Architektur biete, zumindest im Rahmen des Prototyps die den selben Funktionsumfang wie die Nachrichten-basierte Architektur, ist jedoch sehr viel einfacher zu implementieren.

13.3. Design

Der Prototyp besteht aus 3 Bausteinen; Server, API und Client.

Die Bausteine, sowie deren Kommunikation miteinander, werden in den folgenden Kapitel erläutert.

13.3.1. Nachrichten

Alle im System versendeten Nachrichten sind nach dem selben Schema strukturiert, um so unnötige Konvertierungen zu vermeiden.

Eine Nachricht wir entsprechend ihrem Ziel und Funktion benannt.

[Target]_ [Layer]_ [Modul]_ [Funktion]

Teil	Beschreibung
Target	*S* für Server und *C* für Client
Layer	Layername
Modul	Modulname
Funktion	Funktionsname

Der Payload besteht aus dem Meta-Teil, welcher die Collection angibt, sowie dem Data Teil, welcher das neue Objekt und das alte Objekt, falls vorhanden, beinhaltet.

Somit ist implizit eine Mutationsfunktion, und die Referenz auf den Status definiert. Konkret soll dafür die Nachricht das Model referenzieren, damit ist der Name des Datenbankobjekts gemeint, und sowohl das neue Objekt (obj) als auch das referenzierte Objekt (prev) werden mitgesendet.

```
Message = {
    messageName: ""
    meta: {
        model: ""
    }
    data: {
        obj: {}
        prev: {}
    }
```

```
}
```

13.3.2. Backend

Jede über die API eingehende Nachricht wird an den Logik-Layer weitergeleitet.
Der Logik-Layer übernimmt die Verarbeitung, also die Konfliktauflösung und die Verwaltung der Datenhaltung.

Logik

Der Logik-Layer führt die Konfliktauflösung sowie die Verwaltung des Status durch. Es werden vier Nachrichten akzeptiert, welche dem SQL Jargon nachempfunden sind.

Nachrichtname	Beschreibung
S_LOGIC_SM_select	Die Abfrage-Nachricht gibt eine oder mehrere Objekte zurück.
S_LOGIC_SM_create	Die Einfügenachricht erstellt ein neues Objekt und gibt dieses zurück.
S_LOGIC_SM_update	Die Mutationsnachricht aktualisiert ein vorhandenes Objekt.
S_LOGIC_SM_delete	Die Löschnachricht löscht ein vorhandenes Objekt.

Die Resultate werden nach vollständiger Bearbeitung dem Sender der ursprünglichen Nachrichten über eine neue Nachricht mitgeteilt.

Persistenz

Das Verhalten des Singlestate Konzepts ist mit einer Datenbank abbildbar.

13.3.3. API

Die API besteht sowohl aus einem serverseitigem als auch clientseitigem Modul. Nachrichten werden zwischen beiden Modulen ausgetauscht. Im Falle eines Verbindungsunterbruchs werden die Nachrichten zwischengespeichert und bei einer Wiederverbindung zugestellt.

Die Benennung der Nachrichten ist den bekannten Funktionen des HTTP Standards nachempfunden.

Nachrichtname	Beschreibung
S_API_WEB_get	Die Get-Nachricht gibt eines oder alle Objekte einer Collection zurück.
S_API_WEB_put	Ein neues Objekt wird mit der Put-Nachricht erstellt.
S_API_WEB_update	Mit der Update-Nachricht können bestehende Objekte aktualisiert werden.

Nachrichtname	Beschreibung
S_API_WEB_delete	Bestehende Objekte können mit der Delete-Nachricht gelöscht werden.

13.3.4. Frontend

Das Frontend ist der Flux-Architektur nachempfunden. Die beiden Nachrichten können von den Views versendet werden, aktualisieren somit den Store und werden dem Backend übermittelt.

Nachrichtname	Beschreibung
C_PRES_STORE_update	Fügt ein Objekt hinzu oder aktualisiert ein bestehendes.
C_PRES_STORE_delete	Löscht ein bestehendes Objekt.

13.3.5. Datenfluss

Der Datenfluss des Prototypen funktioniert wie in Abbildung 13.1 dargestellt. Zu beachten gilt, dass die gesamte Interkomponenten-Kommunikation asynchron durchgeführt wird.

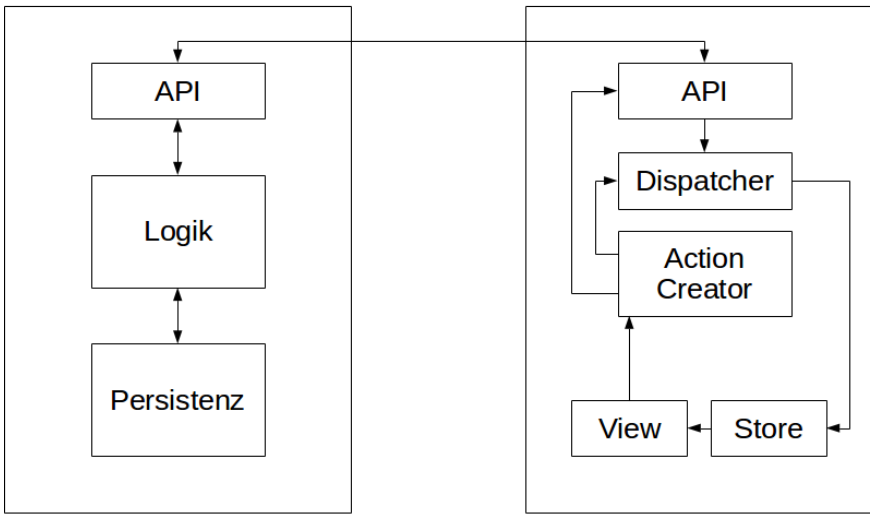
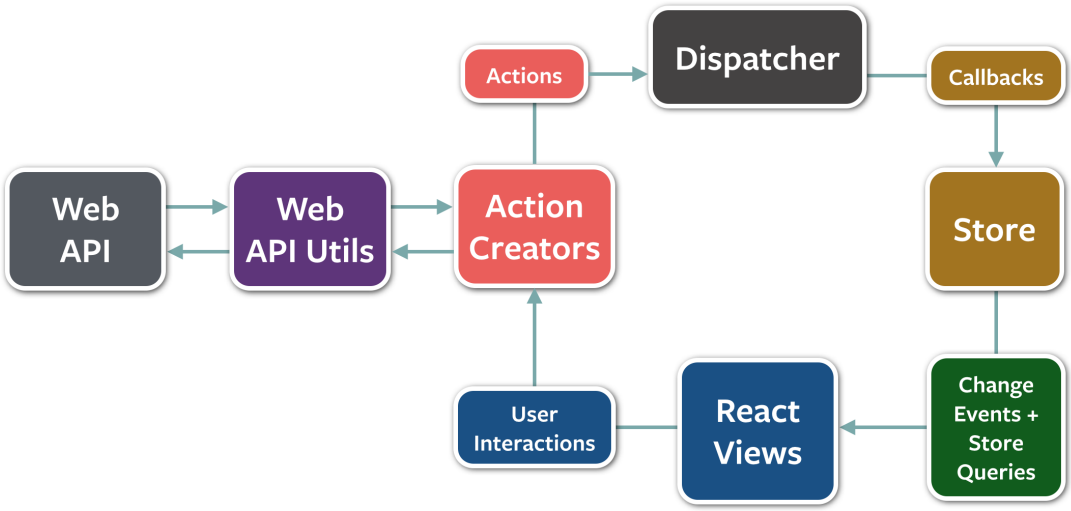


Abbildung 13.1.: Datenflussdiagramm

13.3.6. Flux Architektur

Das Flux Paradigma[5] ist eine Applikationsarchitektur entwickelt von Facebook für React, welche einen unidirektionalen Datenfluss vorgibt.
Daten können nur über eine Aktionen manipuliert werden. Die Views als auch die API können

Aktionen auslösen, und so den Datenbestand mutieren.



Die Verwendung des Dispatchers ermöglicht es, Abhängigkeiten zwischen verschiedenen Stores zentral zu verwalten, da jeder Mutation zwangsweise zuerst von ihm bearbeitet wird.

13.3.7. AMD Pattern

Asyncronous module definition (AMD) ist eine JavaScript API um Module zu definieren und diese zur Laufzeit zu laden. Dadurch können Javascript-lastige Webseiten beschleunigt werden, da Module erst geladen werden, wenn sie gebraucht werden. Weiter werden durch den Loader die Module gleichzeitig geladen, dadurch kann die Bandbreite voll ausgenutzt werden.
Da die Module durch die Vorgabe des Patterns in einzelnen Dateien abgelegt sind, wird eine Kapselung ähnlich wie bei Java erreicht. Das erleichtert die Fehlersuche und erhöht die Verständlichkeit des Programmes drastisch. Auch die Wiederverwendbarkeit der Module wird dadurch erhöht.
Da in jedem Modul die Abhängigkeiten definiert werden müssen, kann während dem Build-Prozess die Abhängigkeiten geprüft werden, um so die Verfügbarkeit aller benötigten Module sicher zu stellen.

13.4. Beispielapplikation

Gem. Aufgabenstellung soll der Prototyp anhand eines passenden Fallbeispiel die Funktionsfähigkeit Zeigen.

Die Beispielapplikation soll eine Ressourcenplan-Software sein. Folgendes soll möglich sein:

1. einen neuen Raum erfassen (Name, Grösse, Anzahl Sitze)
2. einen bestehenden Raum anpassen/löschen
3. einen Termin auf einem Raum Buchen (Name, Zeit&Datum, Kurzbeschreibung, Besucherliste, persönliche Notizen)
4. einen Bestehenden Termin anpassen/absagen

Teil IV.

Implementierung und Testing

14 Prototyp

Dieses Kapitel adressiert die Implementation des Prototypen gemäss den Anroderungen aus Kapitel Analyse.

14.1. Umsetzung

14.2. Technologie Stack

Software	Beschreibung/Auswahlgrund
Grunt	Grunt ermöglicht es dem Benutzer vordefinierte Tasks von der Kommandozeile aus durchzuführen. So sind Build- und Test-Prozesse für alle Benutzer ohne detaillierte Kenntnisse durchführbar. Da Grunt eine sehr grosse Community besitzt und viele Plugins sowie hervorragende Dokumentationen verfügbar, wurde Grunt eingesetzt.
Karma	Karma ist ein Testrunner, der Tests direkt im Browser ausführt. Weiter können automatisch Coverage-Auswertungen durchgeführt werden.
CoffeeScript	CoffeeScript ist ein einfach zu schreibende Sprache die zu JavaScript compiliert. Das generierte JavaScript ist optimiert und ist meist schneller als selbst geschriebenes JavaScript.
RequireJS	RequireJS ermöglicht die Implementierung des AMD Pattern.Dadurch können auch in JavaScript Code-Abhängigkeiten definiert werden. Zusammen mit r.js kann dies bereits zur Compilierzeit geprüft werden. Da weder Backbone noch Django über eine Depencency-Control für JavaScript verfügen, setze ich RequireJS ein.

Software	Beschreibung/Auswahlgrund
ReactJS	ReactJS ist eine Frontend Library die eine starke Modularisierung fordert. Das Paradigma des „Source of Trouth“ verhindert darüber hinaus, dass „komische“ Anzeigefehler auftreten.
FluxifyJS	FluxifyJS ist eine leichtgewichtige Implementierung des Flux Paradigmas. Sie bietet sowohl Stores als auch einen Dispatcher.
SequelizeJS	SequelizeJS ist eine sehr bekannte und weit verbreitete ORM Implementation für Node und Express.
Express	Express ist ein WebFrameowrk für Node. Die weite Verbreitung und ausführliche Dokumentation machen Express zur idealen Grundlage einer Node Applikation.
Socket.io	Socket.io ist eine Implementation des Websocket Standards und erlaubt eine Asynchrone Kommunikation zwischen Client und Server.

14.3. Entwicklungsumgebung

Die Entwicklungsumgebung ist so portabel wie möglich gestaltet. Alle benötigten Abhängigkeiten sind automatisiert installierbar. Die dazu nötigen Befehle sind nachfolgend aufgeführt.

```
> bower install
> npm install
```

14.4. Entwicklung

Sowohl über die API eingehende Nachrichten, als auch interne Nachrichten des Servers werden gleichwertig behandelt.

server/api.coffee

```
me = @
@Socket.on 'message', ( msg ) ->
    me.dispatch msg.messageName, msg.message

flux.dispatcher.register (messageName, message) ->
    me.dispatch messageName, message
```

Express Server:
Statisches Daten -> Frontend /
Socket.io -> /socket.io

Auch das Backend verwendet Fluxify als zentralen MessageBus. Eine Einheitliche Code-Struktur sowie ein besseres Verständnis ist dadurch begünstigt.

Durch den Einsatz von RequireJS Modulen sind diese auch mit Karma direkt im Browser Testbar. So sind statische Analysen über das gesamte Projekt in nur einem Schritt durchführbar.

Stores in the Frontend
client/store.coffee

```
flux.createStore
     id: "prototype_rooms",
    initialState:
        rooms : []

    actionCallbacks:
        C_PRES_STORE_update: ( updater, msg ) ->
            ...
```

Models in the Backend

server/models/*

```
module.exports = (sequelize, DataTypes) ->
    Room = sequelize.define "Room", {
        name: DataTypes.STRING
        description: DataTypes.TEXT

        free: DataTypes.BOOLEAN
        beamer: DataTypes.BOOLEAN
        ac: DataTypes.BOOLEAN
        seats: DataTypes.INTEGER

        image: DataTypes.STRING
    }, {}
```

15
Testing

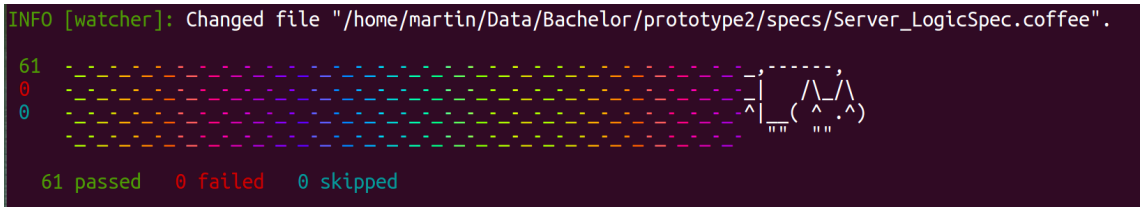


Abbildung 15.1.: Karma Testrunner

Teil V.

Abschluss und Ausblick

16 Review

16.1. Validation

In der nachfolgenden Tabelle sind alle gestellten Ziele gemäss Aufgabenstellung aufgelistet.

Ziel
R1
R2
R3
R4
R5

16.2. Ausblick

17

Fazit und Schlusswort

A
Anhang

A.1. Glossar

ORM
Node

A.2. Aufgabenstellung

A.2.1. Thema

Zeil der Arbeit ist es verschiedene Konfliktlösungsverfahren bei Multi-Master Datenbanksystemen zu untersuchen.

A.2.2. Ausgangslage

Mobile Applikationen (Ressourcen-Planung, Ausleihlisten, etc.) gleichen lokale Daten mit dem Server ab. Manchmal werden von mehreren Applikationen, gleichzeitig, dieselben Datensätze mutiert. Dies kann zu Konflikten führen. Welche Techniken und Lösungswege können angewendet werden, damit Konflikte gelöst werden können oder gar nicht erst auftreten?

A.2.3. Ziele der Arbeit

Das Ziel der Bachelorthesis besteht in der in der Konzeption und der Entwicklung eines lauffähigen Software-Prototypen, welcher mögliche Synchronisations- und Konfliktlösungsverfahren von Clientseitiger und Serverseitiger Datenbank demonstriert. Im Speziellen, soll gezeigt werden, welche Möglichkeiten der Synchronisation beim Einsatz von mobilen Datenbanken (Web-Anwendungen) bestehen, so dass die Clientseitige Datenbank auch ohne Verbindung zum Server mutiert und erst zu einem späteren Zeitpunkt synchronisiert werden kann, ohne dass Inkonsistenzen auftreten. Die Art und Funktionsweise des Software-Prototyp soll in einer

geeigneten Form gewählt werden, so dass verschiedene Synchronisations- und Konfliktlösungsverfahren an ihm gezeigt werden können. Der Software-Prototyp soll nach denen, im Unterricht behandelten Vorgehensweisen des Test Driven Development (TDD) entwickelt werden.

A.2.4. Aufgabenstellung

A1 Recherche:

- Definition der Fachbegriffe
- Erarbeitung der technischen Grundlagen zur Synchronisation von Datenbanken und Datenspeichern

A2 Analyse:

- Analyse der Synchronisationsverfahren und deren Umgang mit Konflikten
- Analyse der Synchronisationsverfahren im Bereich der Web-Anwendungen
- Durchführen einer Anforderungsanalyse an die Software

A3 Konzept:

- Erstellen eines Konzepts der Synchronisation
- Erstellen eines Konzepts der Implementierung zweier ausgewählten Synchronisations-Verfahren

A4 Prototyp:

- Konzeption des Prototypen der die gestellten Anforderungen erfüllt
- Entwickeln des Software-Prototyps
- Implementation zweier ausgewählter Synchronisations- und Konfliktlösungsverfahren

A5 Review:

- Test des Prototyps und Protokollierung der Ergebnisse

A.2.5. Erwartete Resultate

R1 Recherche:

- Glossar mit Fachbegriffen
- Erläuterung der bereits bekannten Synchronisation- und Konfliktlösungs-Verfahren, sowie deren mögliches Einsatzgebiet

R2 Analyse:

- Dokumentation der Verfahren und deren Umgang mit Synchronisation-Konflikten (Betrachtet werden nur MySQL, MongoDB)
- Dokumentation der Verfahren zur Synchronisation im Bereich von Web-Anwendungen (Betrachtet werden nur die Frameworks Backbone.js und Meteor.js)
- Anforderungsanalyse der Software

R3 Konzept:

- Dokumentation des Konzepts der Synchronisation
- Dokumentation der Umsetzung der ausgewählten Synchronisations-Verfahren

R4 Prototyp:

- Dokumentation des Prototypen
- Implementation des Prototypen gemäss Konzept und Anforderungsanalyse
- Implementation zweier ausgewählter Synchronisations- und Konfliktlösungsverfahren

R5 Review:

- Protokoll der Tests des Software Prototypen

A.3. Detailanalyse der Aufgabenstellung

Die Detailanalyse der Aufgabenstellung...

A.3.1. Aufgabenstellung und erwartete Resultate

Recherche:
Es sollen die technischen Grundlagen zur Bearbeitung dieser Thesis zusammengetragen werden. Für das Verständnis wichtige Sachverhalte erläutert und Fachbegriffe erklärt werden.

Erwartet wird ein Glossar, sowie eine Zusammenfassung der bekannten Synchronisations und Konfliktlösungsverfahren, sowie deren Einsatzgebiet.

Analyse:
Eine genauere Betrachtung der ausgewählten Systeme (MySQL, MongoDB, Backbone.js und Meteor.js) zeigt auf, wo die aktuelle Systeme an ihre Grenzen stossen.
Weiter muss eine Anforderungsanalyse für eine Beispielapplikation durchgeführt werden.

Erwartet wird Sowohl die Dokumentation der Synchronisationsverfahren als auch das Ergebnis der Anforderungsanalyse.

Konzept:
Die Erarbeitung und Überprüfung der Umsetzbarkeit neuer Synchronisationskonzepte wird in der Konzeptionsphase gefordert.

Sowohl eine Darstellung der erarbeitete Konzepte, als auch eine Umsetzungsplanung derer ist gefordert.

Prototyp:
Der Prototyp soll anhand eines Beispiels aufzeigen, wo die Stärken und Schwächen eines der Konzepte liegt.

Erwartet wird ein Prototyp der zwei Synchronisations- und Konfliktauflösungsverfahren implementiert.

Review:
Das Review soll eine Retrospektive auf die Erarbeiteten Resultate werfen und kritisch hinterfragen.

Erwartet wird ein Protokoll der durchgeführten Tests.

A.3.2. Aufwandschätzung

Die aus der Projektplanung hervorgehenden Arbeitsschritte müssen geschätzt werden, um eine realistische Terminplanung durchzuführen.

Arbeitsschritt	Aufwand in Stunden
Initialisierung	10
Recherche	45
Analyse	20
Konzeption	80
Prototyp	60
Dokumentation	135
Abgabe	20
Total	**370**

B

Projektmanagement

B.1. Projektplanung

Der vollständige Projektplan ist in der Grafik B.1 dargestellt.

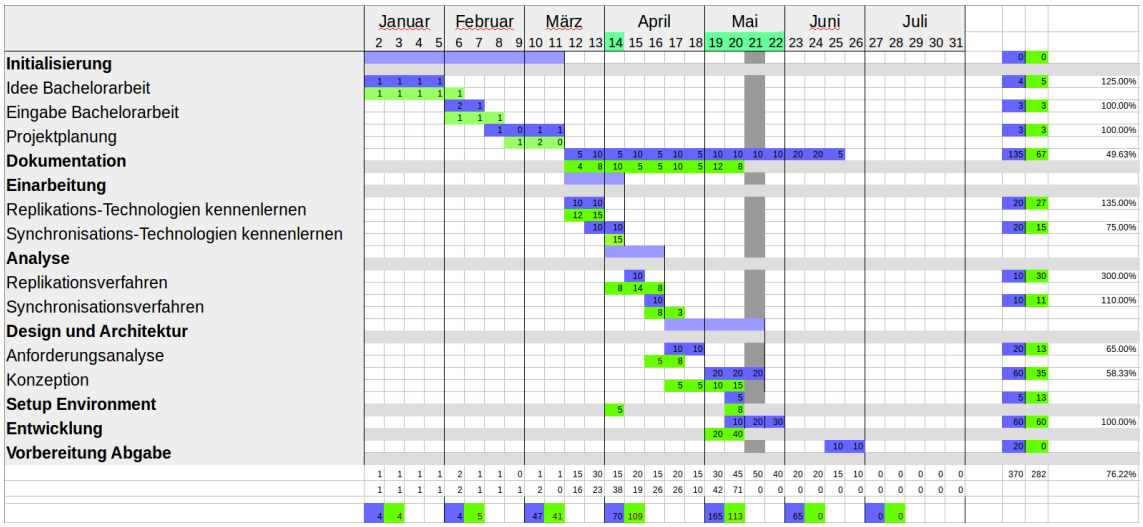


Abbildung B.1.: Projektplan

B.2. Termine

Termin	Datum	Bemerkungen
Kick-Off	18.03.2015	
Design Review	20.05.2015	
Abgabe-Entscheid	06.06.2015	
Abgabe Bachelorthesis	???	

Termin	Datum	Bemerkungen
Abschlusspräsentation	??	

B.3. Dokumentation

Da die Nachvollziehbarkeit von Änderungen in MS Word sehr umständlich ist, habe ich in Betracht gezogen, die Arbeit mit LaTeXzu schreiben.
Da ich jedoch dieses Format sehr unübersichtlich finde habe ich mich stattdessen für Markdown entschieden. Markdown kann mit dem Tool pandoc in ein PDF Dokument konvertiert werden. Darüber hinaus versteht pandoc die Latex-Syntax.

B.4. Versionsverwaltung

Damit einerseits die Daten gesichert und andererseits die Nachvollziehbarkeit von Änderungen gewährleistet ist, verwende ich git.

C

Anforderungsanalyse

C.1. Vorgehensweise

Um eine möglichst allgemein gültige Anforderungsanalyse zu erhalten, werden nur die Anforderungen an den Synchronisationsprozess gestellt, welche für alle beide Fallbeispiele gültig sind.

C.1.1. Use-Cases

Im Nachfolgenden werden alle UseCases aufgelistet die im Rahmen dieser Thesis gefunden wurden.

UC-01 Lesen eines Elements

UseCase	
Ziel	Ein existierendes Objekt wird gelesen.
Beschreibung	Der Benutzer kann jedes Objekt anfordern. Das System liefert das angeforderte Objekt zurück.
Akteure	Benutzer, System
Vorbedingung	Der Benutzer ist im Online-Modus oder Offline-Modus.
Ergebnis	Der Benutzer hat das angeforderte Objekt gelesen.
Hauptszenario	Der Benutzer möchte eine Objekt lesen.
Alternativszenario	-

UC-02 Einfügen eines neuen Elements

UseCase	
Ziel	Ein neues Objekt wird hinzugefügt.

UseCase	
Beschreibung	Der Benutzer kann neue Objekte hinzufügen. Das System liefert das hinzugefügte Objekt zurück.
Akteure	Benutzer, System
Vorbedingung	Der Benutzer ist im Online-Modus oder Offline-Modus.
Ergebnis	Der Benutzer hat ein neues Objekt erfasst.
Hauptszenario	Der Benutzer möchte eine Objekt hinzufügen.
Alternativszenario	-

UC-03 Ändern eines Elements

UseCase	
Ziel	Ein bestehendes Objekt wird mutiert.
Beschreibung	Der Benutzer kann bestehendes Objekte mutieren. Das System liefert das mutierte Objekt zurück.
Akteure	Benutzer, System
Vorbedingung	Der Benutzer ist im Online-Modus oder Offline-Modus.
Ergebnis	Der Benutzer hat ein bestehendes Objekt mutiert.
Hauptszenario	Der Benutzer möchte eine Objekt mutieren.
Alternativszenario	-

UC-04 Löschen eines Elements

UseCase	
Ziel	Ein bestehendes Objekt wird gelöscht.
Beschreibung	Der Benutzer kann bestehendes Objekte löschen.
Akteure	Benutzer, System
Vorbedingung	Der Benutzer ist im Online-Modus oder Offline-Modus.
Ergebnis	Der Benutzer hat ein bestehendes Objekt löschen.
Hauptszenario	Der Benutzer möchte eine Objekt löschen.
Alternativszenario	-

C.1.2. Anforderungen

In diesem Kapitel sind alle funktionalen und nicht-funktionalen Anforderungen die aus den UseCases resultieren ausgeführt.

FREQ01.01 Abfragen eines Objektverzeichnis

Anforderung	
UC-Referenz	UC-01
Beschreibung	Ein Verzeichnis aller Elemente kann abgefragt werden.

FREQ01.02 Abfragen eines bekannten Objekt vom Server

Anforderung	
UC-Referenz	UC-01
Beschreibung	Ein einzelnes Objekt kann von Server abgerufen werden.

FREQ02.01 Senden eines neuen Objekt

Anforderung	
UC-Referenz	UC-02
Beschreibung	Ein einzelnes neues Element kann dem Server zur Anlage zugesendet werden.

FREQ02.02 Abfragen eines neu hinzugefügten Objekt

Anforderung	
UC-Referenz	UC-02
Beschreibung	Das neue angelegte Element wird dem Client automatisch zurückgesendet.

FREQ03.01 Senden einer Objektmutation

Anforderung	
UC-Referenz	UC-03
Beschreibung	Ein Attribut eines existierendes Objekt kann mutiert werden.

FREQ03.02 Senden einer Objektmutation

Anforderung	
UC-Referenz	UC-03
Beschreibung	Ein mehrere Attribute eines existierendes Objekt kann mutiert werden.

FREQ04.01 Löschen eines Objekts

Anforderung	
UC-Referenz	UC-04
Beschreibung	Eine existierendes Objekt kann gelöscht werden.

Anforderung	

FREQ04.01 Lokale Kopie gelesener Objekte

Anforderung	
UC-Referenz	UC-01
Beschreibung	Ein bereits gelesenes Objekt, wird lokal auf dem Client gespeichert.

FREQ05.01 Aufzeichnen der Mutationen

Anforderung	
UC-Referenz	UC-01, UC-02, UC-03, UC-04
Beschreibung	Mutationen werden aufgezeichnet.

FREQ05.02 Übermitteln der Mutationen

Anforderung	
UC-Referenz	UC-01, UC-02, UC-03, UC-04
Beschreibung	Sobald eine Verbindung mit dem Server hergestellt ist, werden die aufgezeichneten Mutationen dem Server übermittelt.

NFREQ01 Übermittlung der Mutationen

Anforderung	
UC-Referenz	UC-01, UC-02, UC-03, UC-04
Beschreibung	Die Übermittlung der Mutationen zum Server darf eine beliebig lange Zeit in Anspruch nehmen.

NFREQ02 Abgelehnte Mutationen

Anforderung	
UC-Referenz	UC-01, UC-02, UC-03, UC-04
Beschreibung	Wenn eine Mutation vom Server abgelehnt wird, wird dem Client die aktuell gültige Version des entsprechenden Objekts übermittelt.

C.1.3. Akzeptanzkriterien

C.1.4. Bewertung der Anforderungen

C.2. Risiken

D
Verzeichnisse

D.1. Quellenverzeichnis

[1] Bernadette Charron-Bost. *Replication: Theory and Practice*. Hrsg. von Fernando Pedone und André Schiper. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. ISBN: 978-3-642-11294-2.

[2] *DB-Engines Ranking*. http://db-engines.com/de/ranking. [Online; accessed 19-March-2015]. 2015.

[3] Mike Elgan. *The hottest trend in mobile: going offline!* http://www.computerworld.com/article/2489829/mobile-wireless/the-hottest-trend-in-mobile--going-offline-.html. [Online, accessed 13-Januar-2015]. 2014.

[4] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf. [Online, accessed 19-Marc-2015]. 2000.

[5] Bill Fisher. *Flux: Actions and the Dispatcher*. http://facebook.github.io/react/blog/2014/07/30/flux-actions-and-the-dispatcher.html. [Online, accessed 10-Mai-2015]. 2014.

[6] Andrew Lipsman. *Major Mobile Milestones in May: Apps Now Drive Half of All Time Spent on Digital*. http://www.comscore.com/Insights/Blog/Major-Mobile-Milestones-in-May-Apps-Now-Drive-Half-of-All-Time-Spent-on-Digital. [Online, accessed 13-Januar-2015]. 2014.

[7] Andreas Meier. *Relationale und postrelationale Datenbanken*. Bd. 7. eXamen.pressSpringerLink. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. ISBN: 978-3-642-05256-9.

[8] *MySQL 5.6 Reference Manual*. http://dev.mysql.com/doc/refman/5.6/en/. [Online, accessed 19-Marc-2015]. 2014.

[9] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. https://bitcoin.org/bitcoin.pdf. 2009.

[10] *Verteilte Algorithmen*. https://www.vs.inf.ethz.ch/edu/HS2014/VA/Vorl.vert.algo12-All.pdf. [Online, accessed 19-March-2015]. 2014.

[11] *Verteilte Systeme*. https://www.vs.inf.ethz.ch/edu/HS2014/VS/slides/VS-Vorl14-all.pdf. [Online, accessed 19-March-2015]. 2014.

D.2. Tabellenverzeichnis

9.1 Attribute Firmen-Kontakt 13
9.2 Attribute Support-Fall 13
9.3 Klassifikation Attribute Kontakt 15
9.4 Klassifikation Attribute Kontakt 15

11.1 Konzept Vergleich Konfliktverhinderung - Konfliktauflösung 29

D.3. Abbildungsverzeichnis

6.1 Verteilung der online verbrachten Zeit nach Platform (Grafik erstellt gemäss der Daten von [6]) 2

10.1 Singlestate 19
10.2 Multistate 20
10.3 Update Transformation 21

13.1 Datenflussdiagramm 34

15.1 Karma Testrunner 40

B.1 Projektplan 48