

# **Evaluation von Synchronisations- und Konfliktlösungsverfahren im Web-Umfeld**

Martin Eigenmann

6. 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

<b>1 Abstract</b>	<b>i</b>
<b>2 Danksagungen</b>	<b>ii</b>
<b>3 Inhaltsverzeichnis</b>	<b>iii</b>
<b>4 Personalienblatt</b>	<b>vi</b>
<b>5 Bestätigung</b>	<b>vii</b>
<b>I Präambel</b>	<b>1</b>
<b>6 Einleitung</b>	<b>2</b>
6.1 Motivation und Fragestellung . . . . .	2
6.2 Aufgabenstellung . . . . .	3
6.3 Abgrenzung der Arbeit . . . . .	3
<b>7 Aufbau der Arbeit</b>	<b>4</b>
<b>II Grundlagen</b>	<b>5</b>
<b>8 Recherche</b>	<b>6</b>
8.1 Fachbegriffe . . . . .	6
8.2 Erläuterung der Grundlagen . . . . .	6
8.3 Replikationsverfahren . . . . .	9
8.4 Synchronisationsverfahren . . . . .	10
<b>III Konzept</b>	<b>12</b>
<b>9 Analyse</b>	<b>13</b>

9.1	Synchronisationsproblem . . . . .	13
9.2	Datenanalyse . . . . .	15
9.3	Datenanalyse der Synchronisationsprobleme . . . . .	16
9.4	Überprüfung der Klassifikation . . . . .	17
<b>10</b>	<b>Konzeptansätze</b>	<b>18</b>
10.1	Synchronisation . . . . .	18
10.2	Datenhaltung . . . . .	19
10.3	Konfliktvermeidung . . . . .	21
10.4	Konfliktauflösung . . . . .	22
<b>11</b>	<b>Konzept Untersuchung</b>	<b>24</b>
11.1	Synchronisation . . . . .	24
11.2	Datenhaltung . . . . .	25
11.3	Konfliktvermeidung . . . . .	27
11.4	Konfliktauflösung . . . . .	28
11.5	Resümee . . . . .	29
<b>12</b>	<b>Leitfaden</b>	<b>30</b>
12.1	Konflikte erlauben . . . . .	30
12.2	Zuständigkeit für Daten . . . . .	30
12.3	Objekte erstellen . . . . .	30
12.4	Sperren . . . . .	31
12.5	Serverfunktionen . . . . .	31
<b>13</b>	<b>Design des Prototypen</b>	<b>32</b>
13.1	Konzeptbestandteile . . . . .	32
13.2	Design . . . . .	32
13.3	Beispielapplikation . . . . .	36
<b>IV</b>	<b>Implementation</b>	<b>37</b>
<b>14</b>	<b>Prototyp</b>	<b>38</b>
14.1	Umsetzung . . . . .	38
14.2	Technologie Stack . . . . .	38
14.3	Entwicklungsumgebung . . . . .	39
14.4	Entwicklung . . . . .	39
<b>15</b>	<b>Testing</b>	<b>42</b>
<b>V</b>	<b>Ausklang</b>	<b>43</b>
<b>16</b>	<b>Review</b>	<b>44</b>
16.1	Validation . . . . .	44
16.2	Ausblick . . . . .	44
<b>17</b>	<b>Fazit und Schlusswort</b>	<b>45</b>
<b>A</b>	<b>Anhang</b>	<b>46</b>

A.1	Glossar . . . . .	46
A.2	Aufgabenstellung . . . . .	46
A.3	Detailanalyse der Aufgabenstellung . . . . .	48
<b>B</b>	<b>Projektmanagement</b>	<b>50</b>
B.1	Projektplanung . . . . .	50
B.2	Termine . . . . .	50
B.3	Dokumentation . . . . .	51
B.4	Versionsverwaltung . . . . .	51
<b>C</b>	<b>Anforderungsanalyse</b>	<b>52</b>
C.1	Vorgehensweise . . . . .	52
C.2	Risiken . . . . .	56
<b>D</b>	<b>Verzeichnisse</b>	<b>57</b>
D.1	Quellenverzeichnis . . . . .	57
D.2	Tabellenverzeichnis . . . . .	58
D.3	Abbildungsverzeichnis . . . . .	58

# 4

## Personalienblatt

Name, Vorname	Eigenmann, Martin
Adresse	Harfenbergstrasse 5
Wohnort	9000 St.Gallen
Geboren	4. Juli 1990
Heimatort	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 Plattform (Grafik erstellt gemäss der Daten von [6])

Angeichts der grossen Verbreitung und Nutzung von Services und Medien im Internet, wird eine unterbrechungsfreie 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 erarbeitete 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<sup>1</sup>), 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.

---

<sup>1</sup><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 erarbeiteten 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 erarbeiteten und implementierten Konzepte der Aufgabenstellung genügen. . .

## **Teil II.**

# **Technische Grundlagen und Erläuterungen**

# 8

## Recherche

### 8.1. Fachbegriffe

Eine Aufführung und dazugehörige 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 <sup>1</sup> 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 <sup>2</sup> 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].

---

<sup>1</sup>In der Literatur oft auch als **DatenBankSystemen** (DBS) oder Informationssystem bezeichnet. [7, pp. 3-4]

<sup>2</sup>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<sup>3</sup> oder Modularität<sup>4</sup> 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<sup>5</sup> 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.

---

<sup>3</sup>Von echter Nebenläufigkeit wird gesprochen, wenn verschiedene Prozesse zur selben Zeit ausgeführt werden. (Multiprozessor)

<sup>4</sup>Modularität beschreibt die Unabhängigkeit und Austauschbarkeit einzelner (Software-) Komponenten. (Auch Lose Kopplung genannt)

<sup>5</sup>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<sup>6</sup>.

#### 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<sup>7</sup> auch auf anderen Teilnehmern mutiert worden sein kann, müssen während des Synchronisationsvorgang<sup>8</sup> 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<sup>9</sup>. Jeder Teilnehmer darf

---

<sup>6</sup>So kann der Backup-Knoten nur Nachts über verfügbar sein, damit der dazwischen liegende Kommunikationsweg Tags über nicht belastet wird.

<sup>7</sup>Zwischen der lokalen Mutation und der Publikation dieser an die übrigen Replikationsteilnehmer, liegt eine beliebige Latenz.

<sup>8</sup>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.

<sup>9</sup>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<sup>10</sup> 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<sup>11</sup> 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 Transactions-Log des Masters wird auf die Slaves kopiert, welche dann die Transaktionen nachführen. Ein Master-Master Betrieb ist nicht vorhergesehen.

---

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

<sup>11</sup>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<sup>12</sup> 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<sup>13</sup>.

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

---

<sup>12</sup>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.

<sup>13</sup>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 entsprechend.

## **Teil III.**

# **Konzeption und Konzeptüberprüfung**

# 9

## Analyse

Bekannte Synchronisationsverfahren betrachten Daten als eine homogene Masse und unterscheiden nicht bezüglich ihrer Beschaffenheit. Konfliktlösungen werden erst auf der Applikationsebene vorgenommen und dann spezifisch auf die Applikationsdaten angewandt. Dieses Kapitel klärt ob eine generelle Unterteilung der Daten möglich ist, und ob dies einen Nutzen mit sich bringt. Die Analyse ist auf einer abstrahierten Stufe durchgeführt und blendet bewusst alle technischen Aspekte aus. Daten sind nur aus der Sicht eines Benutzers betrachtet. Alle für den Benutzer nicht sichtbaren Daten werden nicht zu Analyse herangezogen.

### 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, durch eine andere parallel laufende Transaktion, ändern können.

Um ein Konzept zur Verhinderung und Lösung Synchronisationskonflikten zu erarbeiten, muss zuerst eine Analyse der zu synchronisierenden Daten durchgeführt werden.

Die Datenanalyse wird mit Bezug auf die zwei im Folgenden aufgeführten Problemstellungen durchgeführt. Beide Problemstellungen sind so gewählt, dass sie kombiniert einen möglichst allgemeinen Fall abdecken können und so die Überprüfung aussagekräftig wird.

#### 9.1.1. Synchronisation von Kontakten

In vielen Anwendungsszenarien müssen Kontaktdaten zwischen verschiedenen Systemen und Plattformen synchronisiert werden. In diesem Beispiel wird ein verallgemeinerte Verwendungszweck in einer Unternehmung beleuchtet.

Eine Firma „VIP Customer Care“ betreibt eine zentrale Kontaktdatenbank an ihrem Hauptsitz. Darin sind alle Kunden mit Namen, Adresse, Telefonnummern und Email-Adressen erfasst. Zusätzlich ist es für jeden Anwender möglich persönliche Notizen zu einem Kontakt zu erfassen. Die Anwender müssen jederzeit Kontaktdaten abfragen, anpassen oder neu erfassen können,

ohne dafür mit dem zentralen Kontaktdatenbank verbunden zu sein. Gerade in wenig entwickelten oder repressiven Ländern ist eine Verbindung nicht immer voraussetzbar und somit eine Off-Line Funktionalität notwendig.

Die Anpassungen werden dann zu einem späteren Zeitpunkt abgeglichen und somit in die zentrale Kontaktdatenbank synchronisiert.

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

Tabelle 9.1.: Attribute Firmen-Kontakt

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

### 9.1.2. Synchronisation eines Service Desks

Ein anderes Anwendungsszenario bezieht sich auf die Synchronisation von Ablaufdaten in einem definierten Umfeld. So werden in vielen Fällen auch Daten von Abläufen und Vorgängen synchronisiert. Abhängigkeiten (zeitlich sowie inhaltlich) zwischen den Daten sind stark ausgeprägt und müssen von der Business-Logik überprüft werden.

In vielen IT-Organisationen kommt irgend eine Form eines Service-Desks zum Einsatz. Typischerweise werden Arbeitszeiten, Aufwendungen und Tätigkeiten direkt auf einen Support-Fall gebucht. Auch die Kommunikation mit dem Kunden wird hauptsächlich über den Service-Desk geführt. Das Erfassen dieser Einträge soll jederzeit möglich sein, auch ohne Verbindung zum Service-Desk.

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
<b>Titel</b>	Titel des Support-Falls.
<b>Beschreibung</b>	Fehler/Problembeschreibung des Tickets.
<b>Anmerkungen</b>	Alle Antworten von Technikern und Kunden. Eine Antwort des Technikers kann als FAQ-Eintrag markiert werden.
<b>Arbeitszeit</b>	Alle erfassten und aufgewendeten Stunden für den Support-Fall.

Attribut	Beschreibung
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 und 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.

### 9.3.1. Synchronisation von Kontakten

Der Name eines Kontakts ist der primäre Identifikator eines Kontakts. Er alleine zeigt dem Benutzer an, um welchen Kontakt es sich handelt. Der Name ändert sich nur in Extremfällen und gibt somit den Kontext des Kontaktes an.

Die Attribute Adresse, Email und Telefon sind deshalb allesamt abhängig vom Namensattribut. Diese Attribute sind also kontextuell abhängig vom Identifikator. Nur solange der Name nicht geändert wurde, ist die Übernahme von Anpassungen an den Attributen Adresse, Email und Telefon sinnvoll.

Das Attribut pNotes hingegen ist völlig unabhängig, da es nur vom Verfasser gelesen und geschrieben werden kann. Ob die darin enthaltenen Informationen also zum Kontext passen, liegt alleine in der Verantwortung des Autors und muss vom System nicht weiter beachtet werden.

Die Attribute Name, Adresse, Email und Telefon können von allen Benutzern des Systems jederzeit verändert werden. Jeder Benutzer ist gleichberechtigt, niemand wird bevorzugt. Aus diesem Grund sind diese Attribute als gemeinsame Daten klassifiziert. Das Attribut pNotes hingegen ist für jeden Benutzer exklusiv editierbar und einsehbar. Jeder Benutzer sieht und bearbeitet also nur seine eigene Version des Attributs.

Obwohl alle Attribute ein sehr unterschiedliches Erscheinungsbild aufweisen, sind sie Textfelder. Zwar können diesen Textfeldern Formate wie Telefonnummer oder Adresse hinterlegt werden, sie sind aber auf der Ebene der Daten trotzdem nur Text-Felder.

Alle im Fallbeispiel gezeigten Attribute lassen sich klassifizieren und einem Datentyp zuordnen. Die Klassifikation repräsentiert die Struktur sowie die Art des Zugriffs auf die Daten auf einem hohen Abstraktionslevel. Es gibt keine nicht klar klassifizierbaren Attribute.



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

### 9.3.2. 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

Die durchgeführte Datenanalyse der Synchronisationsprobleme der beiden Fallbeispiele zeigt, dass sowohl die Klassifikation nach Struktur, als auch die Klassifikation nach Art durchführbar und repräsentativ ist. Es kann klar zwischen der Klassifikation exklusiv, gemeinsam und dynamisch unterschieden werden. Auch sind die beiden Struktur-Klassen kontextbezogen und kontextunabhängig sind allgemeingültig und ermöglichen eine Repräsentation der Abhängigkeiten zwischen den verschiedenen Attributen.

Die beiden weiteren vorgeschlagenen Arten-Klassen, statisch und temporär, sind nicht eindeutig genug, um eingesetzt werden zu können. Daten die mit der Art temporär klassifiziert werden könnten, können auch durch die Klasse exklusiv klassifiziert werden. Die Klasse exklusiv ist sogar noch genereller und eine Unterscheidung der Klassen nach der Gültigkeitsdauer der klassifizierten Daten bieten keinen Mehrwert. Die zweite Arten-Klasse statisch ist ebenso repräsentativer durch der Klasse gemeinsam abgedeckt.

Die gefundenen Klassen können also auf Daten angewendet werden und repräsentieren diese auch auf dem gewünschten Abstraktionslevel.

# 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. Dabei kann zwischen den beiden Vorgehensweisen Nachrichten basiert und Objekt basiert unterschieden werden.

#### 10.1.1. Unterschieds basiert

Zur Durchführung einer Unterschieds basierten Synchronisation muss sowohl die Mutationsfunktion als auch der Status, auf welchen sie angewendet wird, bekannt sein. Beide Informationen zusammen werden als eine Einheit betrachtet und als Nachricht bezeichnet.

Eine Status-Mutation wird 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. Jede einzelne Änderung am Datenbestand wird durch eine einzige Nachricht repräsentiert. Die Nachricht wird direkt bei der Änderung generiert und Server übermittelt, sobald eine Verbindung hergestellt werden konnte.

#### 10.1.2. Objekt basiert

Der Abgleich der Datenbestände wird durchgeführt sobald eine Verbindung mit dem Server besteht. Dann werden alle Objekte, die geändert und noch nicht synchronisiert wurde, vollständig dem Server übermittelt. Jedes Objekt wird in einer Nachricht zusammen mit der Referenz, des zuletzt vom Server erhaltenen Objekts, übermittelt.

## 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 Anwendung der später eingegangene Nachricht  $N_3$  auf  $S_2$  führt zum ungültigen Status  $S'_3$  und löst damit einen Synchronisationskonflikt aus.

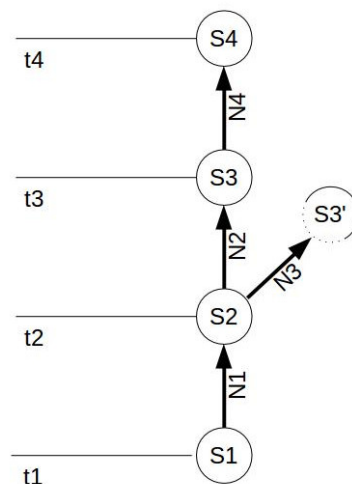


Abbildung 10.1.: Singlestate

Die Konfliktauflösung 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  $S_{3,1}$ .

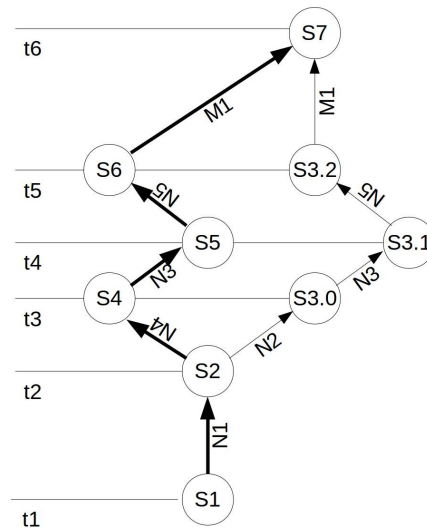


Abbildung 10.2.: Multistate

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 deshalb 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.

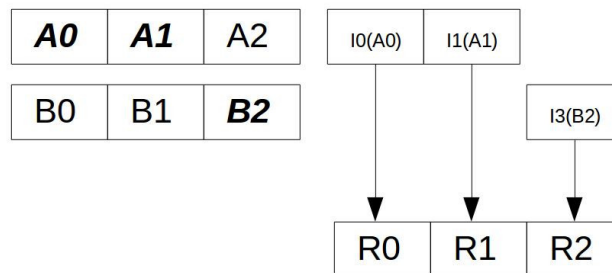


Abbildung 10.3.: Update Transformation

#### 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.

#### 10.3.3. Serverfunktionen

Funktionen werden nur auf dem Server ausgeführt. Diese Funktionen sind auf dem Client nicht verfügbar und können deshalb nur ausgeführt werden, sobald eine Verbindung zum Server besteht. Dieses Konzept wird im Bereich der RPC verwendet.

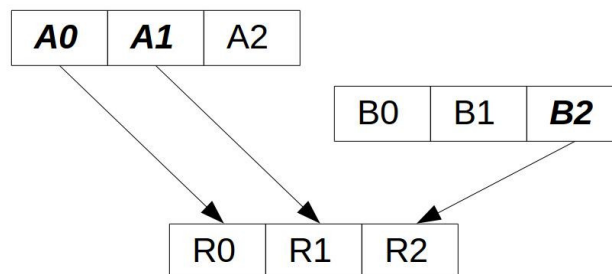
## 10.4. Konfliktauflösung

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 folgenden 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. geschätzte Zusammenführung

Wenn zwei oder mehr Mutationen auf einen Status angewendet werden sollen, wird die wahrscheinlich beste Mutation verwendet. Dazu wird die Richtigkeit jeder einzelnen Mutation geschätzt. Diese Schätzungen kann entweder mit statischen Analyse oder mit neuronalen Netzen umgesetzt werden.

### 10.4.3. 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 numerische 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. Synchronisation

Zur Übermittlung von Informationen über Mutationen am Datenbestand, wird eine Form der Markierung benötigt. Die mutierten Daten müssen markiert werden, damit sie, sobald die Synchronisierung durchgeführt werden soll, übermittelt werden können.

#### 11.1.1. Unterschieds basiert

Die Unterschieds basierte Synchronisation zeichnet direkt bei der Mutation eines Objekts die an den Server zu sendende Nachricht auf und speichert diese in einer Messagequeue zur späteren Synchronisation ab. Sobald also die Synchronisation ausgelöst wird, werden alle aufgezeichneten Nachrichten, in der gleichen Reihenfolge wie bei der Aufzeichnung, dem Server zugestellt.

```
synchronize(): ->
    for Message in @MessageQueue
        sendToServer(Message)

mutateObject(Mutation, referenceObject): ->
    @Messagequeue.add(Mutation, referenceObject)
```

#### 11.1.2. Objekt basiert

Bei der Objekt basierten Synchronisation wird direkt bei der Mutation eines Objekts, dessen *dirty* Flag gesetzt. Bei der Auslösung der Synchronisation werden alle Objekte, welche dieses Flag gesetzt haben, dem Server übermittelt und das *dirty* Flag wieder entfernt.



```

synchronize(): ->
    for Element in @Store.getAll()
        if Element.dirty
            sendToServer(Element)
            Element.dirty = false

mutateObject(Element): ->
    obj = @Store.get(Element)
    obj.attrs = Element.attrs
    obj.dirty = true

```

## 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

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 rückwirkend unveränderbaren Status. Auch eine Versionierung und somit ein wahlfreier Zugriff auf alle Stati ist implementierbar.

Die Implementation auf konzeptioneller Stufe ist dabei wenig anspruchsvoll.

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().canResolveConflict Message

```

```

        @State().apply Message
    else
        break

```

Die Funktionen `canResolveConflict` 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 Status 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.

### 11.2.2. Multistate

Die Multistate Implementation unterscheidet sich insbesondere darin, dass das Annehmen einer Nachricht und das Auflösen des Konflikts voneinander unabhängig sind.

Beim Eingehen einer neuen Nachricht wird ebenfalls die Funktion `addMessage()` aufgerufen. Die Funktion **StateTree**( $t$ ) gibt 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.tryToResolveConflict

```

## Performance

Da bei jeder schreibenden Operation der gesamte Statusbaum neu aufgebaut wird, weist diese Implementation 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.

## 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
```

### 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.

### 11.3.3. Serverfunktionen

Serverfunktionen werden nur auf dem Server ausgeführt. Dafür wird eine Nachricht generiert, die es erlaubt, Funktionen direkt auf dem Server auf zu rufen. Neben dem Funktionsnamen, können auch Argumente mitgegeben werden.

```
composeMessage(FunctionName, Args): ->

    Message.type = RPC
    Message.name = FunctionName
    Message.args = Args

    return Message
```

## 11.4. Konfliktauflösung

### 11.4.1. Zusammenführung

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

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

    NewState = new State

    for AttrName, Attribut in current
        if reference[AttrName] is valid[AttrName]
            NewState[AttrName] = Attribut
        else
            break

    return NewState
```

### 11.4.2. geschätzte Zusammenführung

Zur Auflösung von Konflikten mittels der geschätzten geschätzten Zusammenführung wird eine Distanzfunktion benötigt. Diese Distanzfunktion ermittelt den Abstand zur optimalen Lösung und wendet dann die Mutation mit dem geringsten Abstand an.

```
resolveConflict (valid, reference, average): ->

    NewState = new State
    Distances = new DistanceCalculator()

    for update in reference
```

```
Distances.add update

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

return NewState
```

## 11.5. Resümee

In diesem Kapitel sind die einzelnen Teile des Konzeptes kondensiert zusammengefasst und hinsichtlich der Praxistauglichkeit bewertet.

Die Unterschieds basierte Synchronisation ist sehr granular und flexibel einsetzbar. Die Logik des Synchronisierens ist vollständig von der Datenhaltung entkoppelt und ermöglicht einen sehr flexiblen Einsatz auch in bereits bestehenden Projekten. Dahingegen benötigt die Objekt basierte Synchronisation eine Anpassung an der clientseitigen Datenhaltung.

Die Multistate Datenhaltung erlaubt zwar die zeitliche Entkoppelung von Synchronisation und Konfliktauflösung, garantiert jedoch keine Isolation, keine Atomarität und auch keine Konsistenz. Vor allem die Tatsache, dass wiederholte Abfragen, nicht das selbe Resultat zurückliefern, birgt grosse Risiken im Betrieb.

Sowohl die wiederholbare Transaktion, als auch die geschätzte Zusammenführung lösen schwierige 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 sind gut einsetzbar und schwächen das Synchronisationsproblem deutlich ab.

# 12

## Leitfaden

Dies ist ein Set von Konventionen und Richtlinien für die Synchronisation von Daten im Web-Umfeld basierend auf den Ergebnissen aus der Analyse und Auswertung der Beispieldaten. Der Wert von Software ist direkt gekoppelt an die Qualität ihrer Codebasis. Nicht nur die Fehleranfälligkeit, sondern auch die Wartbarkeit steigt mit der Verwendung von nicht durchdachten Designs.

Diese Regeln helfen Probleme bei der Synchronisation zu reduzieren und gleichzeitig die Wartbarkeit zu erhöhen.

### 12.1. Konflikte erlauben

Die Applikation soll die Möglichkeit des Auftretens von Konflikten vorsehen. So sollen benutzerfreundliche Fehlermeldungen generiert werden, die den Benutzer darauf hinweist, dass von ihm bearbeitete Daten und Informationen nicht übernommen werden konnten. Konflikte sollen darüber hinaus aufgezeichnet und für Analysen gespeichert werden. Nicht übernommene Daten werden so gespeichert und gehen nicht verloren.

### 12.2. Zuständigkeit für Daten

Wenn immer möglich, sollen Daten nur einem Benutzer zugewiesen sein. Somit ist Verwaltung und Veränderung der Daten nur einem Benutzer möglich. Synchronisationskonflikte entfallen so fast vollständig.

### 12.3. Objekte erstellen

Wenn immer möglich sollen neue Objekte erstellt werden statt bestehenden zu mutieren. Zusätzliche Informationen werden dazu in neuen Objekten, entsprechen referenziert, hinzugefügt,

## **12.4. Sperren**

Das Setzen von Sperren erlaubt es vorübergehend alle anderen Mutationen zu verbieten. Dadurch kann ein Benutzer konfliktfrei Änderungen durchführen.

Dabei wird bei Betreten des Editiermodus eines Objekts, dieses auf dem Server für alle anderen Benutzer gesperrt. Diese nun, bis der Bearbeiter das Objekt speichert und damit wieder freigibt, nur noch lesend darauf zugreifen.

## **12.5. Serverfunktionen**

Das Verwenden von Serverfunktionen beschränkt die Ausführung dieser nur auf den Zeitraum, in dem eine Verbindung zum Server besteht. Dadurch werden kritische Mutationen zeitnah vom Server verarbeitet und die Wahrscheinlichkeit für Konflikte sinkt drastisch.

# 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 Richtlinien aus dem Kapitel Leitfaden genügen.

### 13.1. Konzeptbestandteile

Die Konventionen und Richtlinien des Kapitels Leitfaden berücksichtigend müssen nicht alle im Konzept erarbeiteten Ansätze umgesetzt werden.

Die Datenübermittlung wird Unterschieds basiert durchgeführt und serverseitig mit einem auf dem Konzept des Singlestate basierenden Datenspeichers komplettiert. Zur Konfliktvermeidung wird sowohl die Update Transformation, wiederholbare Transaktionen als auch Serverfunktionen verwendet. Bezüglich der Konfliktauflösung wird nur die Zusammenführung umgesetzt, da Konflikte explizit erlaubt sind.

### 13.2. Design

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



Die API-Komponente steht für sich alleine, obschon sie von Server und Client direkt angesprochen wird und implementationisch in eine Server und eine Client Part geteilt ist. Weiter übernimmt die API-Komponente den Nachrichtenaustausch zwischen Server und Client.



Die Bausteine, sowie deren Interaktion miteinander, wird in den folgenden Kapitel genauer erläutert.

### 13.2.1. Backend

Das Backend ist in drei Schichten unterteilt (Api, Logik und Persistenz), um so eine möglichst grosse Separation of Concerns (SoC) zu erreichen. Die Kommunikation zwischen diesen Schichten findet nur über Nachrichten statt.

Jede über die API eingehende Nachricht wird an den Logik-Layer weitergeleitet. Der Logik-Layer übernimmt dann die Verarbeitung und leitet dann die Resultate der Persistenz-Schicht weiter.

### 13.2.2. API

Die API stellt wenn immer möglich einen ständigen Kommunikationskanal zwischen Server und Client her. Der serverseitige Part der API kann sowohl broadcast Nachrichten als auch an nur einen Client gerichtet Nachrichten verwenden. Die API verfügt jedoch über keine MessageQueue und Nachrichten die nicht beim ersten Versuch zugestellt werden können, gehen verloren.

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

Tabelle 13.1.: Nachrichten Server-API

Nachrichtename	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.
S_API_WEB_delete	Bestehende Objekte können mit der Delete-Nachricht gelöscht werden.
S_API_WEB_execute	Die Execute-Nachricht führt eine Serverfunktion aus.

### Logik

Der Logik-Layer führt die Konfliktauflösung sowie die Verwaltung des Status durch. Es werden vier Nachrichten akzeptiert, welche dem SQL Jargon nachempfunden sind. Die Resultate werden nach vollständiger Bearbeitung dem Sender der ursprünglichen Nachrichten über eine neue Nachricht mitgeteilt.

Tabelle 13.2.: Nachrichten Server-Logik

Nachrichtename	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.
S_LOGIC_SM_execute	Die Ausführnachricht löst die Ausführung einer Serverfunktion aus.

### Persistenz

Die Persistenz muss nur einen einzigen Status verwalten. Das Verhalten des Singlestate Konzepts ist mit einer Datenbank abbildbar. Der referenzierte Status wird jeweils in der Nachricht vom Client mitgeliefert. Ein wahlfreier Zugriff auf vergangene Stati ist deshalb nicht notwendig.

### 13.2.3. Frontend

Das Frontend ist auf der Flux-Architektur aufbauen. Die beiden Nachrichten können von den Views versendet werden, aktualisieren den Store und werden vom Client-API Teil verarbeitet.

Tabelle 13.3.: Nachrichten Client

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

### 13.2.4. Nachrichten

Nachrichten können entweder eine Serverfunktion aufrufen oder eine Mutation durchführen. Der Payload besteht aus dem Meta-Teil, welcher den Objektnamen oder die Serverfunktion bezeichnet, sowie dem Data-Teil, welcher die Mutationsfunktion oder die Argumente der Serverfunktion beinhaltet.

Die Mutationsfunktion wird implizit durch die Übermittlung vom neuen Status (obj) und dem vorhergehenden Status (prev) umgesetzt.

```
Message = {
  messageName: ""
  meta: {
    model: ""
    function: ""
  }
}
```

```

    }
    data: {
      obj: {}
      prev: {}
      args: {}
    }
  }
}

```

### 13.2.5. Datenfluss

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

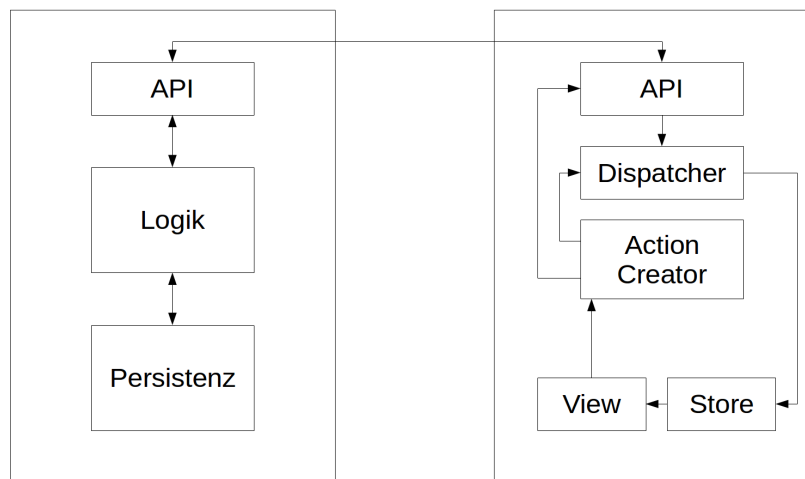
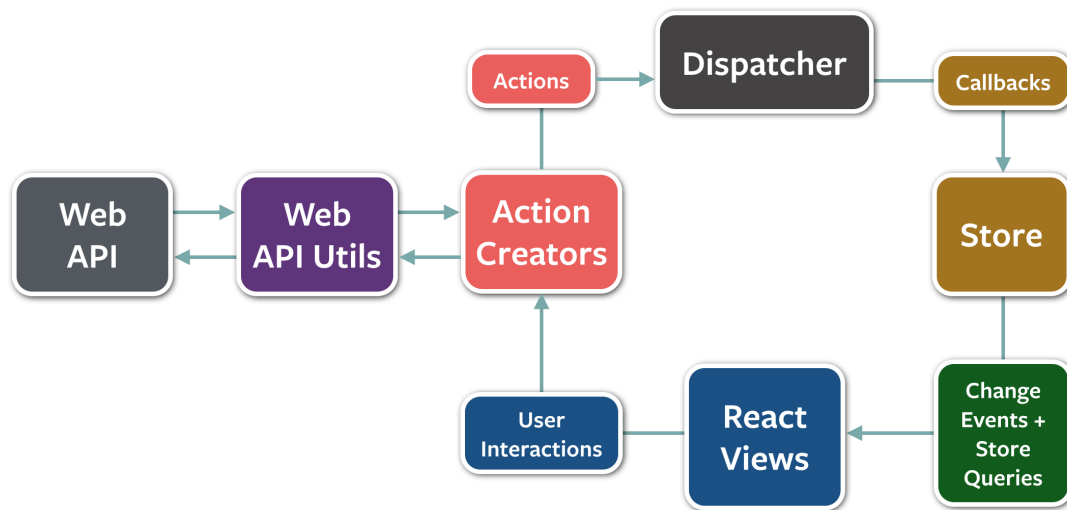


Abbildung 13.1.: Datenflussdiagramm

### 13.2.6. 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.

### 13.2.7. AMD Pattern

Asynchronous 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.

## 13.3. 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 Anforderungen aus Kapitel Analyse.

### 14.1. Umsetzung

### 14.2. Technologie Stack

Software	Beschreibung/Auswahlgrund
<b>Grunt</b>	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.
<b>Karma</b>	Karma ist ein Testrunner, der Tests direkt im Browser ausführt. Weiter können automatisch Coverage-Auswertungen durchgeführt werden.
<b>CoffeeScript</b>	CoffeeScript ist eine einfach zu schreibende Sprache, die zu JavaScript kompiliert. Das generierte JavaScript ist optimiert und ist meist schneller als selbst geschriebenes JavaScript.
<b>RequireJS</b>	RequireJS ermöglicht die Implementierung des AMD Pattern. Dadurch können auch in JavaScript Code-Abhängigkeiten definiert werden. Zusammen mit r.js kann dies bereits zur Compilerzeit geprüft werden. Da weder Backbone noch Django über eine Dependency-Control für JavaScript verfügen, setze ich RequireJS ein.

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

Die Implementation des Konfliktverhinderungsverfahrens „Wiederholbare Transaktion“ sowie der Konfliktauflösungsverfahren „Zusammenführung“, „kontextbezogene Zusammenführung“ sowie die traditionelle Synchronisation

```
_repeatable: (data, db_obj, new_obj, prev_obj, attr) ->
  if new_obj[attr]?
    data[attr] = db_obj[attr] + (new_obj[attr] - prev_obj[attr])

_combinging: (data, db_obj, new_obj, prev_obj, attr) ->
  if new_obj[attr]?
    data[attr] = if new_obj[attr] is prev_obj[attr]
      then db_obj[attr] else new_obj[attr]

_traditional: (data, db_obj, new_obj, prev_obj, attr) ->
  if new_obj[attr]?
    data[attr] = if prev_obj[attr] is db_obj[attr]
      then new_obj[attr] else db_obj[attr]

_contextual: (data, db_obj, new_obj, prev_obj, attr, context) ->
  if new_obj[attr]?
    data[attr] = if prev_obj[context] is db_obj[context]
      then new_obj[attr] else db_obj[attr]
```

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

Die Testrunner Suite Karma erlaubt es Programmcode direkt im Browser zu testen und statische Analyse darüber anzufertigen.

```
INFO [watcher]: Changed file "/home/martin/Data/Bachelor/prototype2/specs/Server_LogicSpec.coffee".  
61  
0  
0  
61 passed 0 failed 0 skipped
```

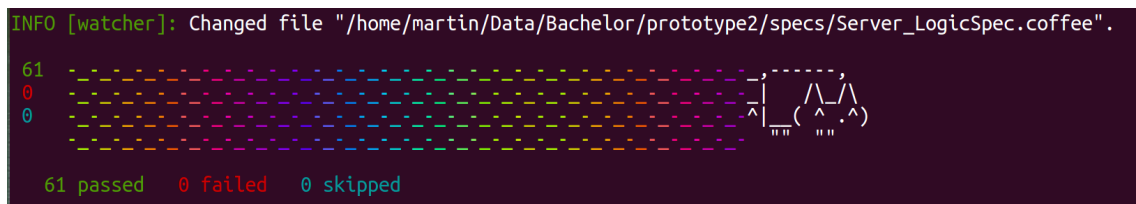
A screenshot of a terminal window with a dark purple background. The text is displayed in various colors: green for 'INFO', red for 'failed', and blue for 'skipped'. The output shows a file change notification, a series of colored dashes representing test progress, and a final summary line indicating that 61 tests passed, 0 failed, and 0 were skipped. A small ASCII art logo is visible on the right side of the terminal output.

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



# Anhang

## A.1. Glossar

ORM  
Node  
RPC

## 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
<b>Total</b>	<b>370</b>



# Projektmanagement

## B.1. Projektplanung

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

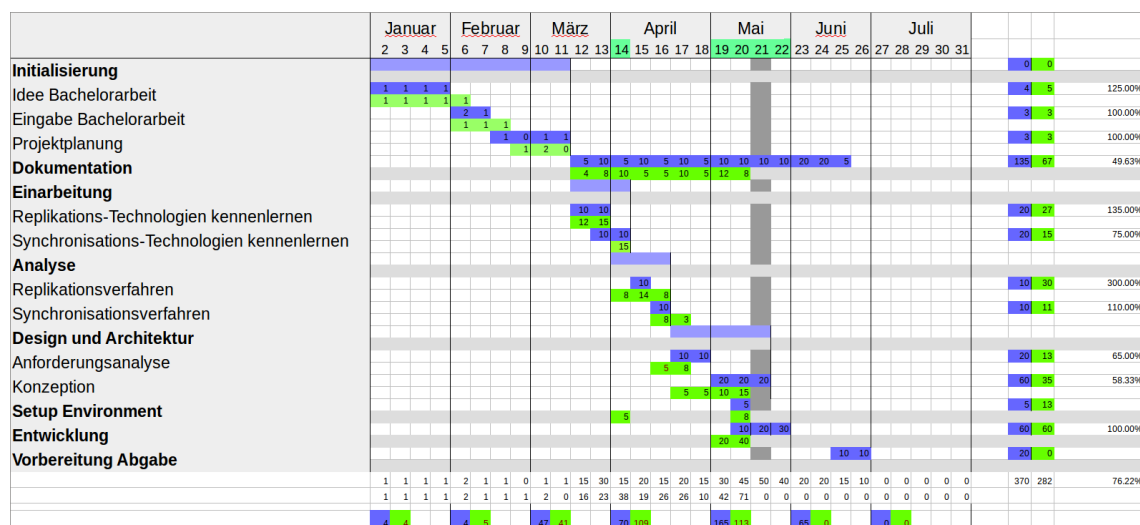


Abbildung B.1.: Projektplan

## 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  $\text{\LaTeX}$  zu 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.



# 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	
<b>Ziel</b>	Ein existierendes Objekt wird gelesen.
<b>Beschreibung</b>	Der Benutzer kann jedes Objekt anfordern. Das System liefert das angeforderte Objekt zurück.
<b>Akteure</b>	Benutzer, System
<b>Vorbedingung</b>	Der Benutzer ist im Online-Modus oder Offline-Modus.
<b>Ergebnis</b>	Der Benutzer hat das angeforderte Objekt gelesen.
<b>Hauptscenario</b>	Der Benutzer möchte eine Objekt lesen.
<b>Alternativscenario</b>	-

#### UC-02 Einfügen eines neuen Elements

UseCase	
<b>Ziel</b>	Ein neues Objekt wird hinzugefügt.

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

### UC-03 Ändern eines Elements

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

### UC-04 Löschen eines Elements

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

## 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	
<b>UC-Referenz</b>	UC-01
<b>Beschreibung</b>	Ein Verzeichnis aller Elemente kann abgefragt werden.

#### **FREQ01.02 Abfragen eines bekannten Objekt vom Server**

<b>Anforderung</b>	
<b>UC-Referenz</b>	UC-01
<b>Beschreibung</b>	Ein einzelnes Objekt kann von Server abgerufen werden.

#### **FREQ02.01 Senden eines neuen Objekt**

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

#### **FREQ02.02 Abfragen eines neu hinzugefügten Objekt**

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

#### **FREQ03.01 Senden einer Objektmutation**

<b>Anforderung</b>	
<b>UC-Referenz</b>	UC-03
<b>Beschreibung</b>	Ein Attribut eines existierendes Objekt kann mutiert werden.

#### **FREQ03.02 Senden einer Objektmutation**

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

#### **FREQ04.01 Löschen eines Objekts**

<b>Anforderung</b>	
<b>UC-Referenz</b>	UC-04
<b>Beschreibung</b>	Eine existierendes Objekt kann gelöscht werden.

---

**Anforderung**

---

**FREQ04.01 Lokale Kopie gelesener Objekte**

---

**Anforderung**

---

<b>UC-Referenz</b>	UC-01
<b>Beschreibung</b>	Ein bereits gelesenes Objekt, wird lokal auf dem Client gespeichert.

---

**FREQ05.01 Aufzeichnen der Mutationen**

---

**Anforderung**

---

<b>UC-Referenz</b>	UC-01, UC-02, UC-03, UC-04
<b>Beschreibung</b>	Mutationen werden aufgezeichnet.

---

**FREQ05.02 Übermitteln der Mutationen**

---

**Anforderung**

---

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

---

**NFREQ01 Übermittlung der Mutationen**

---

**Anforderung**

---

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

---

**NFREQ02 Abgelehnte Mutationen**

---

**Anforderung**

---

<b>UC-Referenz</b>	UC-01, UC-02, UC-03, UC-04
<b>Beschreibung</b>	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**





# 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](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-Vor114-all.pdf>. [Online, accessed 19-March-2015]. 2014.

## D.2. Tabellenverzeichnis

9.1	Attribute Firmen-Kontakt . . . . .	14
9.2	Attribute Support-Fall . . . . .	14
9.3	Klassifikation Attribute Kontakt . . . . .	17
9.4	Klassifikation Attribute Kontakt . . . . .	17
13.1	Nachrichten Server-API . . . . .	33
13.2	Nachrichten Server-Logik . . . . .	34
13.3	Nachrichten Client . . . . .	34

## D.3. Abbildungsverzeichnis

6.1	Verteilung der online verbrachten Zeit nach Plattform (Grafik erstellt gemäss der Daten von [6]) . . . . .	2
10.1	Singlestate . . . . .	19
10.2	Multistate . . . . .	20
10.3	Update Transformation . . . . .	21
13.1	Datenflussdiagramm . . . . .	35
15.1	Karma Testrunner . . . . .	42
B.1	Projektplan . . . . .	50