Zürcher Hochschule für Angewandte Wissenschaften
Studiengang Informatik
Entwickeln für Handheld

Zürcher Hochschule
für Angewandte Wissenschaften
zh aw School of Engineering

Evaluation von Synchronisations- und Konfliktlösungsverfahren im Web-Umfeld

Martin Eigenmann

26. Juni 2015

ZUSAMMENFASSUNG

Das Abgleichen von Daten zwischen Server- und Client-Applikation (Synchronisation) stellt eine spezielle Herausforderung, gerade dann dar, wenn mehre Clients gleichzeitig Daten mutieren. Durch das gleichzeitige Mutieren von Daten kann es so zu Inkonsistenzen kommen. Speziell mobile Applikationen für Smartphones oder Tablets dürfen nicht von einem ständigen Kommunikationskanal mit dem Server ausgehen uns müssen deshalb mit potentiellen Konflikten umgehen können. Diese Arbeit handelt von der Analyse und Erstellung von Konzepten und Lösungsansätzen zur Abschwächung des schweren Dantensynchronisationsproblems im Web-Umfeld.

Bekannte Synchronisationsverfahren verlangen entweder eine andauernde Verbindung zwischen dem Server und allen Clients, schränken den Funktionsumfang (Client kann nur lesend auf die Daten zugreifen) ein, oder garantieren keine zeitliche Aktualität der Daten auf den Clients.
Anhand einer umfassenden Analyse der Beschaffenheit und Struktur der Daten unter Zuhilfenahme von Fallbeispielen, welche den allgemeinen Anwendungsfall repräsentativ wiedergeben, werden Konzepte zur Synchronisation, Datenhaltung, Konfliktvermeidung und Konfliktauflösung erarbeitet, bewertet und überprüft. Basierend auf den erarbeiteten Lösungsansätze ist eine Überprüfung der Umsetzbarkeit jedes einzelnen Konzepts durchgeführt, um so allfällige Schwächen und bis dahin unentdeckte Probleme festzustellen.
Aufgrund der gewonnenen Erkenntnissen wird ein Leitfaden zur Abschwächung des Synchronisationsproblem erarbeitet, welcher die wichtigsten zu beachtenden Punkte wieder gibt. Eine Anforderungsanalyse an das Fallbeispiel “Synchronisation einer Kontaktdatenbank” mit Use Cases und funktionalen sowie nicht funktionalen Anforderungen setzen die funktionsbezogenen Leitplanken für den Prototypen. Unter Betrachtung der Vor- und Nachteile jedes einzelnen Konzepts und unter Berücksichtigung des Leitfadens, wird für die Entwicklung des Prototyps die geeignetsten Konzepte ausgewählt, konzipiert und in Form eines Proof of Conzept implementiert. Sowohl das Konzept der Zusammenführung (Synchronisation auf Ebene von Attributen und nicht ganzer Datensätze) sowie der kontextbezogenen Zusammenführung (Beachtung der Abhängigkeiten zwischen Attributen eines Datensatzes) erlauben zusammen mit der unterschiedsbasierten Synchronisation (Übermittlung nur des Deltas der Änderung) und dem expliziten vorsehen von Synchronisationskonflikten eine deutliche Reduktion der Komplexität.
Über die gesamte Erstellungsphase hinweg, stehen dabei Techniken zur Erstellung guter Software, wie Test Driven Development und Domain Driven Architecture, im Mittelpunkt. Neben der nachrichtenbasierte Kommunikation zwischen Client und Server, kommunizieren auch die einzelnen Layer und Komponenten, client- wie auch server-seitig über Nachrichten und somit asynchron. Die so erreichte Entkoppelung und implizite Schnittstel-

lendefinition der Software gegen innen und aussen erlaubt darüber hinaus ein verständliches und nachvollziehbares Testing.
Mit dem Prototyp des implementierten Fallbeispiels kann gezeigt werden, dass nachrichtenorientierte Kommunikation zusammen mit der Einführung des Konzepts einer kontextuellen Beziehung zwischen den einzelnen Attributen innerhalb eines Datensatzes, die Komplexität des Synchronisationsproblems reduziert.

INHALTSVERZEICHNIS

TEIL I

EINLEITUNG UND ABGRENZUNG

1 EINLEITUNG

1.1 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. [Lip14]

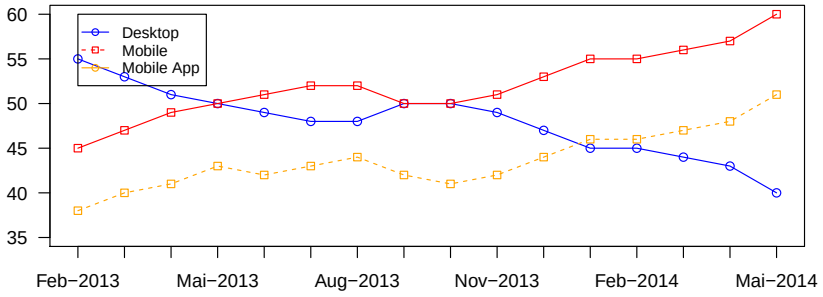


Abbildung 1: Verteilung der online verbrachten Zeit nach Platform (Grafik erstellt gemäss der Daten aus [Lip14])

Angesichts 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 [Elg14]

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 können, 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 durchzuführen.

1.2 AUFGABENSTELLUNG

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

1.3 ABGRENZUNG DER ARBEIT

Grosse Anbieter von Web-Software wie Google und Facebook arbeiten intensiv an spezifischen Lösungen für ihre Produkte. Zwar werden in Vorträgen, 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 werden 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

2

AUFBAU DER ARBEIT

Teil i - Präambel

Im ersten Kapitel Einleitung werden die Daten-Synchronisations-Probleme mit mobilen Endgeräten angedeutet und die daraus resultierende Motivation für diese Arbeit dargelegt. Weiter wird die Abgrenzung dieser Arbeit erörtert.
Schliesslich wird im Kapitel Aufbau der Arbeit die Struktur sowie die Eigenleistungen betont.

Teil ii - Grundlagen

Das Kapitel Recherche setzt sich mit den Grundlagen, Fachbegriffen und bekannten Verfahren zur Synchronisation und Replikation auseinander. Bekannte Verfahren zur verteilten Ausführung von Programmen, synchronen und asynchronen Replikation von Datenbanken sowie Synchronisationsverfahren werden umrissen.

Teil iii - Konzept

In diesem Teil der Thesis werden die Grundlagen geschaffen um darauf basierend Konzepte zu erstellen. Zwei Fallbeispiele für die Synchronisation von Daten werden eingeführt. Es wird darauf basierend eine Analyse der zu synchronisierenden Daten erstellt und überprüft ob eine sinnvolle Klassifikation der Daten durchführbar ist um so das Synchronisationsproblem abzuschwächen.
Basierend auf der Analyse der Daten, werden Konzeptansätze erarbeitet und anschliessend auf ihre Anwendbarkeit hin überprüft. Die sich daraus ergebenen Erkenntnisse werden im Leitfaden übersichtlich dargestellt. Die Ansätze des Leitfadens berücksichtigend wird der Aufbau des Prototypen (Design des Prototypen) konzipiert.

Teil iv - Implementation

Dieser Teil beschäftigt sich mit der Implementierung des Prototypen unter Verwendung der TDD Methode. Ein spezielles Augenmerk wird dem Testing und der verwendeten Frameworks gewidmet.

Teil v - Ausklang

In diesem Teil der Thesis wird evaluiert, ob die erarbeiten und implementierten Konzepte der Aufgabenstellung genügen, welche Limitationen sie aufweisen und welche Fragestellungen offen geblieben sind.

TEIL II

TECHNISCHE GRUNDLAGEN UND ERLÄUTERUNGEN

3 RECHERCHE

3.1 FACHBEGRIFFE

Eine Aufführung und dazugehörende Erklärung, der für das Verständnis der Arbeit notwendigen Fachbegriffe befindet sich im Anhang unter dem Kapitel "Glossar".

3.2 ERLÄUTERUNG DER GRUNDLAGEN

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

3.2.1 *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 verwaltenden Daten selbst. Ein DBMS muss 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. [Dbe]

3.2.2 *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.

1 In der Literatur oft auch als **D**aten**B**ank**S**ystem (DBS) oder Informationssystem bezeichnet. [Mei10, pp. 3-4]
2 Bekannt als ACID-Prinzip [Mei10, pp.105] umfasst es **A**tomicity, **C**onsistency, **I**solation und **D**urability.

3.2.3 *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. [Ethb]

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.

3.2.4 *Verteilte Algorithmen*

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

3.2.5 *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. [CB10, p. 19]

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.

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 2-Phase und 3-Phase-Commit ermöglichen darüber hinaus auch das synchrone Editieren von Objekten auf allen Replikationsteilnehmern. [CB10, 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 Synchronisationsvorgangs[8] eventuell auftretenden Konflikte aufgelöst werden.

3.2.6 *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 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. [Nak09]

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

3.3 REPLIKATIONSVERFAHREN

3.3.1 *MySQL*

Das Datenbanksystem MySQL unterstützt sowohl 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[Mys]:

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.

3.3.2 *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.

3.4 SYNCHRONISATIONSVERFAHREN

3.4.1 *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

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

REST-API lesen sowie ein einzelnes Objekt schreibend an die REST-API senden.
Der REST-Standard[Fie00] verlangt dass für die Übermittlung eines Objekt immer der vollständige Status übermittelt wird. So wird bei lesendem und schreibenden Zugriff immer das 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, welche konkurrierende Version bei der Synchronisation 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 in 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 das Objekt vom Client erneut angefordert, um so die aktuellste Version des Objekts zu beziehen.

3.4.2 *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 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.

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.

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 dem entsprechen.

TEIL III

KONZEPTION UND KONZEPTÜBERPRÜFUNG

Um die Komplexität des zu analysierenden und zu lösenden Problems deutlich zu reduzieren, wird eine Aufteilung dessen durchgeführt. Zu Beginn wird das Problem selbst sowie die zu übertragenden Daten anhand zweier Beispiele beleuchtet und analysiert. In einem zweiten Schritt werden verschiedene Konzeptansätze zu den Teilbereichen Synchronisation, Datenhaltung, Konfliktvermeidung und Konfliktauflösung erarbeitet und anschliessen auf ihre Anwendbarkeit hin untersucht. Die Ergebnisse dieser Untersuchung sind im "Leitfaden" auf die wichtigsten Findings hin reduziert. Auf diesen Hinweisen aufbauend, wird anschliessend ein Design für den Prototypen erarbeitet.

4 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 generalisierte Unterteilung der Daten möglich ist, und ob dies, bezüglich der Abschwächung des Synchronisationsproblems, 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.

4.1 SYNCHRONISATIONSPROBLEM

Generell liegt ein potentieller 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.

4.1.1 *Synchronisation von Kontakten*

In vielen Anwendungsszenarien müssen Kontaktdaten zwischen verschiedenen Systemen und Plattformen synchronisiert werden. In diesem Beispiel wird ein verallgemeinerter Verwendungszweck in einer Unternehmung beleuchtet.
Eine Firma 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 der zentralen Kontaktdatenbank verbunden zu sein. Gerade in wenig entwickelten oder repressiven Ländern ist eine Verbindung nicht immer voraussetzbar und somit eine Offline 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 1: Attribute Firmen-Kontakt

Attribut	**Beschreibung**
Name	Der gesamte Name (Vor-, Nach- ,Mittelname und Titel) der Kontakt-Person.
Adresse	Die vollständige Postadresse der Firma oder Person.
Email	Die aktive Email-Adresse des Kontakts.
Telefon	Die aktive Telefonnummer des Kontakts.
pNotes	Persönliche Bemerkungen zum Kontakt. Nur der Autor einer Notiz, kann diese bearbeiten oder lesen.

4.1.2 *Syncrhonisation 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.

Tabelle 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.

4.2 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.

4.2.1 *Klassifikation nach Art*

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

- *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.

4.2.2 *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 Betrachten zusätzlicher Informationen, nicht mehr Wissen, bezüglich des ursprünglichen 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 und ist deshalb als kontextbezogen zu klassifizieren.

4.2.3 *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.

4.3 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.

4.3.1 *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 Textfelder.
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 3: Klassifikation Attribute Kontakt

Attribut	**Struktur**	**Art**	**Typ**
Name	Unabhängig	gemeinsam	textuell
Adresse	Abhängig (Name)	gemeinsam	textuell

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

4.3.2 *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 4: Klassifikation Attribute Kontakt

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

4.4 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, anwendbar 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 mit der Klasse exklusiv, mit gleichwertiger Aussagekraft klassifiziert werden. Die Klasse exklusiv ist sogar noch genereller und eine Unterscheidung der Klassen nach der Gültigkeitsdauer der klassifizierten Daten bietet keinen Mehrwert. Die zweite Arten-Klasse statisch ist ebenso besser durch der Klasse gemeinsam repräsentiert.
Die gefundenen Klassen können also auf Daten angewendet werden und repräsentieren diese auch auf dem gewünschten Abstraktionslevel.

5 KONZEPTANSÄTZE

5.1 SYNCHRONISATION

Die grundlegende Idee bei der Synchronisation liegt darin, den Zustand des Servers und des Clients, bezüglich der Daten, identisch zu halten. Der Zustand der Daten kann 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 unterschiedsbasierte und objektbasierte Synchronisation unterschieden werden. Beide Ansätze sind im Folgenden beschrieben.

5.1.1 *Unterschiedsbasierte Synchronisation*

Zur Durchführung einer unterschiedsbasierten Synchronisation muss sowohl die Mutations-Funktion 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 dem Server übermittelt, sobald eine Verbindung hergestellt wird.

5.1.2 *Objektbasierte Synchronisation*

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

5.2 DATENHALTUNG

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

5.2.1 *Singlestate*

Ein Singlestate System erlaubt, nach dem Vorbild traditioneller Datenhaltungssysteme, zu jedem Zeitpunkt nur einen einzigen gültigen Zustand.
Eingehende Nachrichten N enthalten sowohl die Mutations-funktion als auch eine Referenz auf welchen Status, 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 2 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.

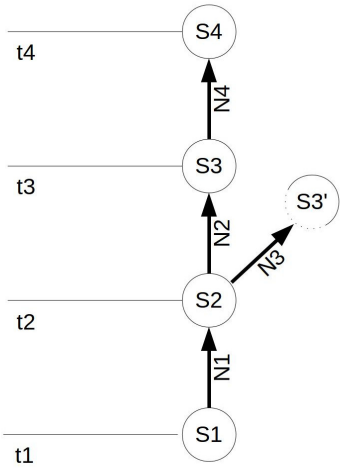


Abbildung 2: Singlestate

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.

5.2.2 *Multistate*

Dieses Konzept arbeitet genau so wie ein Singlestate System mit eingehenden Nachrichten. Ein Multistate System erlaubt jedoch zu jedem Zeitpunkt beliebig viele gültige Zustände. Zu jedem Zeitpunkt ist jedoch immer nur ein Zustand aktiv.
Dieses Verhalten wird dadurch erreicht, dass Zustände rückwirkend eingefügt werden können. Wenn zum Zeitpunkt t_4 und t_6 der gültige Zustand des Systems zum Zeitpunkt t_3 erfragt wird, muss nicht notwendigerweise der identische Zustand zurückgegeben werden.

In der Abbildung 3 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 $S_{3.1}$.

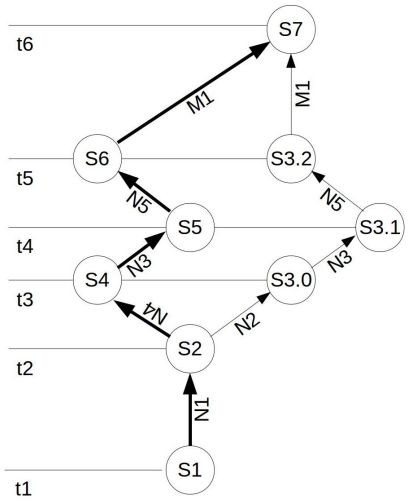


Abbildung 3: Multistate

Der Zeitpunkt an welchem die Nachricht N_4 verarbeitet ist, wird nun mit 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 $S_{3.0}$ aktiv. Danach ist der Status S_4 aktiv.
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 obwohl der Referenzstatus nicht identisch ist. Eine Mutations-Funktion kann also auf jeden, vom referenzierten Status abstammenden Status angewendet werden, solange sie in keinem Konflikt damit steht. 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 die, der Nachricht N_2 und Status $S_{3.2}$ alle Informationen ausser die, 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 aktiv zu definieren ist, wird mit einer 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 Kon-

fliktauflösung darf sehr greedy entscheiden, da Fehlentscheide zu einem anderen Zeitpunkt korrigiert werden können.

5.3 KONFLIKTVERMEIDUNG

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

5.3.1 *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 4 zeigt, muss nicht das gesamte Objekt aktualisiert werden und es wird so ermöglicht die Konfliktauflösung granularer durch zu führen.

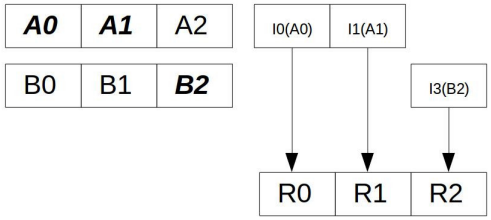


Abbildung 4: Update Transformation

5.3.2 *Wiederholbare Transaktion*

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

5.3.3 *Serverfunktionen*

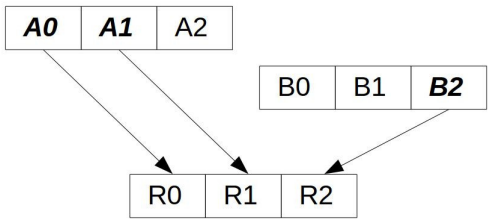
Funktionen werden nur auf dem Server ausgeführt. Diese Serverfunktionen sind auf dem Client nicht verfügbar und können deshalb nur ausgeführt werden, sobald eine Verbindung zum Server besteht. Dieses Konzept entspringt dem Konzept von Remote Procedure Call und garantiert Konfliktfreiheit zur Ausführungszeit.

5.4 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 der Konfliktauflösung eine entscheidende Rolle einnehmen, muss diese Beschaffenheit mit einbezogen werden.

5.4.1 *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.



5.4.2 *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.

5.4.3 *Kontextbezogene Zusammenführung*

Für jedes, als kontextbezogen klassifiziertes Attribut, muss ein Kontextattribut, also ein anderes Attribut des selben Objekts, definiert werden. Bei der Konfliktauflösung wird das Attribut nur dann übernommen, wenn sich das Kontextattribut nicht geändert hat.

5.4.4 *Vergabelungs-Funktion*

Die Vergabelungs-Funktion kann nur zusammen mit dem Dantenhaltungskonzept Multistate eingesetzt werden.
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 dafür 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.

6 KONZEPTUNTERSUCHUNG

6.1 SYNCHRONSATION

Zur Aufzeichnung der 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.

6.1.1 *Unterschiedsbasiert*

Die unterschiedsbasierte 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)
```

6.1.2 *Objektbasiert*

Bei der objektbasierten 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
```

6.2 DATENHALTUNG

Eine sehr elegante Form der Datenhaltung ist die Messagequeue. Der aktuell gültige Status ist durch die Anwendung aller in der Messagequeue enthaltenen Nachrichten auf den initialen Status erreichbar.
Der wahlfreie Zugriff auf jeden beliebigen Status 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.

6.2.1 *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* gibt den Status zum Zeitpunkt t zurück. Falls kein Zeitpunkt angegeben ist, wird der neueste zurückgegeben.
Die MessageQueue wird durch das Statusobjekt 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().canResolvConflict Message
            @State().apply Message
        else
            break
```

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

Verbesserung

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

6.2.2 *Multistate*

Die Multistate Implementation unterschiedet sich insbesondere darin, dass das Annehmen einer Nachricht und das Auflösen des Konflikts zeitlich voneinander unabhängig sind.
Beim Eingehen einer neuen Nachricht wird ebenfalls die Funktion *addMessage* aufgerufen. Die Funktion *StateTree* 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.tryToResolvConflict
```

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.

6.3 KONFLIKTVERMEIDUNG

Die Konfliktvermeidung zielt darauf ab, Konflikte gar nicht erst entstehen zu lassen. Dazu müssen entweder funktionale Einschränkungen oder erhöhte Komplexität des Vorgangs hingenommen werden. Die Konzepte sind im Folgenden erläutert.

6.3.1 *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
```

6.3.2 *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.

6.3.3 *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
```

6.4 KONFLIKTAUFLÖSUNG

Die Konfliktauflösung wird erst ausgeführt, wenn Konflikte auftreten. Im eine übersichtlichere Implementation zu ermöglichen, übernimmt die Konfliktauflösung jedoch auch die Konflikterkennung.

6.4.1 *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
```

6.4.2 *Kontextbezogene Zusammenführung*

Zur Implementation der kontextbezogenen Zusammenführung, muss der Kontext auf Ebene der Attribute definiert sein. Nur Attribute bei welchen sich der Kontext nicht änderte, werden übernommen.

```
resolveConflict (current, contextFor): ->

    NewState = new State

    for AttrName, Attribut in current
        if contextFor[AttrName].didNotChange
            NewState[AttrName] = Attribut
        else
            break

    return NewState
```

6.4.3 *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
```

6.5 KONKLUSION

In diesem Kapitel sind die einzelnen Teile des Konzeptes kondensiert zusammengefasst und hinsichtlich der Praxistauglichkeit bewertet.
Die unterschiedsbasierte 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 objektbasierte 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. Der Singlestate ist deshalb deutlich besser zur Datenhaltung geeignet.
Sowohl die wiederholbare Transaktion, als auch die geschätzte Zusammenführung lösen schwierige Konflikte. Da die 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.

6.6 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 Bewertung der erarbeiteten Konzepte.
Diese fünf Regeln sollen das sehr schwere Synchronisationsproblem im Web-Umfeld abzuschwächen und somit die Komplexität der Software zu reduzieren.

6.6.1 *Unterschiedsbasierte Synchronisation*

Mutationen am Datenbestand sollen in der chronologischen Reihenfolge ihres Auftretens zwischengespeichert werden, um sie anschliessend in genau der gleichen Reihenfolge auf dem Server anwenden zu können.

6.6.2 *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.

6.6.3 *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.

6.6.4 *Objekte erstellen*

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

6.6.5 *Lock*

Das setzten von Sperren erlaubt es vorübergehend alle anderen Mutationen zu verbieten. Dadurch kann ein Benutzer konfliktfrei Änderungen durchführen. Dabei wird beim Aufrufen des Editiermodus eines Objekts, dieses auf dem Server für alle anderen Benutzer gesperrt. Diese können nun, bis der aktuelle Bearbeiter das Objekt speichert und damit wieder freigibt, nur noch lesend auf das Objekt zugreifen.

6.6.6 *Serverfunktionen*

Das Verwenden von Serverfunktionen beschränkt die Ausführung dieser nur auf den Zeitraum, in dem eine Verbindung zum Server besteht. Dadurch werden kritische Mutationen synchron vom Server verarbeitet und Konfliktfreiheit wird somit für diese eine Operation garantiert.

7 DESIGN DES PROTOTYPEN

In diesem Kapitel wird ein Prototyp entworfen, der, der Konklusion aus dem Kapitel [Konzept Untersuchung] und den Richtlinien aus dem Kapitel Leitfaden genügt.

7.1 DESIGNENTSCHEIDUNGEN

Die im Leitfaden vorgeschlagenen Lösungsansätze sollen in das Design des Prototypen mit einfliessen. Die beiden Konventionen "Zuständigkeit für Daten" und "Lock" bedingen jedoch eine Benutzerverwaltung und finden daher keinen Eingang.
Die Datenübermittlung wird basierend auf den Erkenntnissen aus den vorhergehenden Kapiteln, Unterschieds basiert durchgeführt und serverseitig mit einem, auf dem Konzept des Singlestate basierenden, Datenspeicher komplettiert. Dieses Nachrichten basierte Kommunikationskonzept wird geradezu perfekt durch die Flux Architektur umgesetzt. Jede Benutzerinteraktion mit dem Client, erzeugt eine neue Nachricht. Daten verändernde Nachrichten können so auf einfache Art und Weise auch dem Server übermittelt werden. Um die Modulare Struktur, welche vom Flux vorgegeben wird zu komplettieren wird das AMD Pattern verwendet. Dadurch können Module und deren Abhängigkeiten gleichermassen für Server und Client definiert werden.
Serverfunktionen sind über Nachrichten mit dem darin enthaltenen Argumenten aufrufbar und über aufgetretene Konflikte wird der Client mit einer entsprechenden Nachricht informiert. Die Konvention "Objekte erstellen" besitzt im umzusetzenden Fallbeispiel nur einen geringen Einfluss, da die Datenstruktur bereits auf ein einziges Objekt pro Kontakt festgelegt ist.

7.2 AUSWAHL SYNCHRONISATIONS- UND KONFLIKTAUFLÖSUNGSVERFAHREN

Zur Konfliktvermeidung werden die beiden erarbeiten Konzepte Update Transformation und Serverfunktionen umgesetzt. Bezüglich der Konfliktauflösung wird nur die Zusammenführung umgesetzt, da einerseits Konflikte explizit erlaubt sind und andererseits nur die Zusammenführung garantiert fehlerfreie Resultate liefert.

7.3 DESIGN

Der Prototyp besteht aus den drei Bausteinen: Server, API und Client.

Die API-Komponente steht für sich alleine, obschon sie sowohl im Server als auch im Client direkt eingebettet ist. Die API-Komponente selbst ist aufgeteilt in einen Server- und Client-Teil und stellt den Austausch der Nachrichten zwischen Server und Client sicher.
Die Bausteine und deren Interaktion miteinander, werden in den folgenden Kapitel genauer erläutert.

7.3.1 *Datenfluss*

Der Datenfluss des Prototypen ist wie in der Abbildung 5 dargestellt, nur unidirektional. Daraus ergibt sich auch, dass die gesamte Interkomponenten-Kommunikation, vom Client bis hin zum Server, asynchron durchgeführt wird.

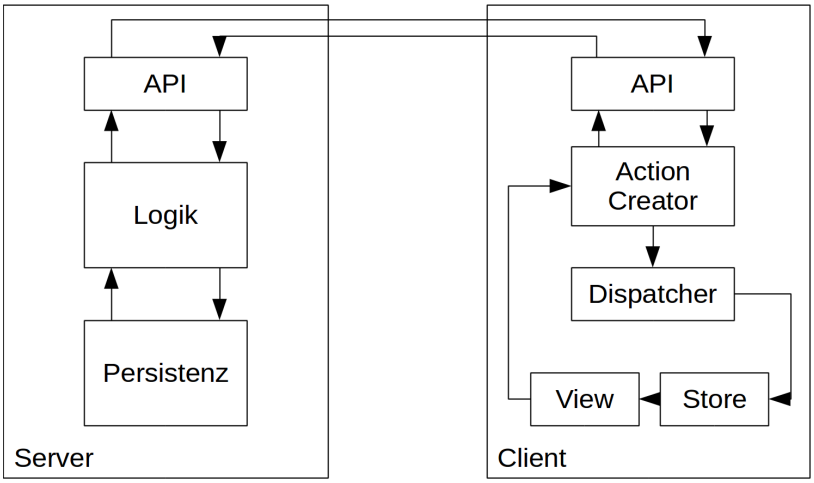


Abbildung 5: Datenflussdiagramm von Server und Client

7.3.2 *Nachrichten*

Jede versendete Nachricht enthält neben einem Nachrichtennamen und den zu versendenden Daten, auch Meta-Informationen. In den Meta-Informationen ist der Name des mutierten Objekts oder der Name der Serverfunktion eingetragen.

Neben der des gesamten veränderten Elements wird auch das zuvor aktive Element in den Daten übermittelt. Implizit ist mit der Übermittlung der alten und neuen Daten die Mutationsfunktion definiert.

7.3.3 *Backend*

Das Backend ist in drei Schichten, Api, Logik und Persistenz, unterteilt, 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 die Resultate der Persistenz-Schicht weiter.

7.3.4 *API*

Die API stellt wenn immer möglich einen ständigen Kommunikationskanal zwischen Server und Client her. Der serverseitige Part der API kann sowohl Broadcastnachrichten als auch an nur einen Client gerichtete Nachrichten versenden. Die API verfügt jedoch über keine MessageQueue und Nachrichten die nicht beim ersten Versucht zugestellt werden können, gehen verloren.
Die Benennung der Nachrichten ist den bekannten Funktionen des HTTP Standards nachempfunden. So ist die tatsächliche Implementation der API maximal flexibel und trotzdem immer noch standardkonform.

Tabelle 5: Nachrichten Server-API

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.
S_API_WEB_delete	Bestehende Objekte können mit der Delete-Nachricht gelöscht werden.
S_API_WEB_execute	Eine Serverfunktion kann mit der Execute-Nachricht ausgeführt werden.

Logik

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

Tabelle 6: Nachrichten Server-Logik

Nachrichtname	**Beschreibung**
S_LOGIC_SM_select	Die Select-Nachricht gibt eine oder mehrere Objekte zurück.
S_LOGIC_SM_create	Die Create-Nachricht erstellt ein neues Objekt und gibt dieses zurück.
S_LOGIC_SM_update	Die Update-Nachricht aktualisiert ein vorhandenes Objekt.
S_LOGIC_SM_delete	Die Delete-Nachricht löscht ein vorhandenes Objekt.
S_LOGIC_SM_execute	Die Execute-Nachricht löst die Ausführung einer Serverfunktion aus.

Persistenz

Das Verhalten des Singlestate Konzepts ist mit einer Datenbank abbildbar. Der referenzierte Status wird jeweils in der Nachricht vom Client, vollständig mitgeliefert. Das Implementieren eines wahlfreien Zugriffs auf vergangene Stati ist deshalb nicht notwendig.

7.3.5 *Frontend*

Die beiden Nachrichten können von den Views versendet werden, aktualisieren den Store und werden vom Client-API Teil verarbeitet. Nur diese beiden Nachrichten aktualisieren den Store des Clients.

Tabelle 7: Nachrichten Client

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

7.3.6 *Interaktionen*

Die Ausgestaltung des Ablaufs der Interaktionen zwischen Client und Server ist für die drei wichtigsten Fälle im Folgenden aufgeführt.

Initiale Synchronisation

Die Abbildung 6 illustriert den Ablauf der initialen Synchronisation. Sobald sich der Client mit dem Server verbunden hat, wird der initiale Datenbestand des Servers an den Client übermittelt. Dazu wird jedes Element des Servers einzeln in einer Nachricht an den Client gesendet. Der Vorgang ist abgeschlossen sobald eine Kopie aller Elemente versendet wurde.

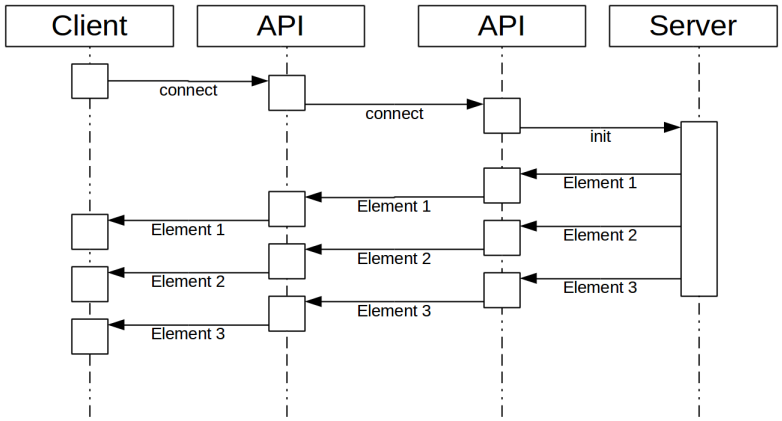


Abbildung 6: Ablauf der initiale Synchronisation

Synchronisation

Beim durchführen lokaler Änderungen werden die dazugehörenden Nachrichten bereits an die API weitergereicht und dort zur Weiterleitung an den Server zwischengespeichert.
Sobald eine Verbindung mit dem Server besteht werden diese zwischengespeicherten Nachrichten, wie in Abbildung 7, an den Server übermittelt.

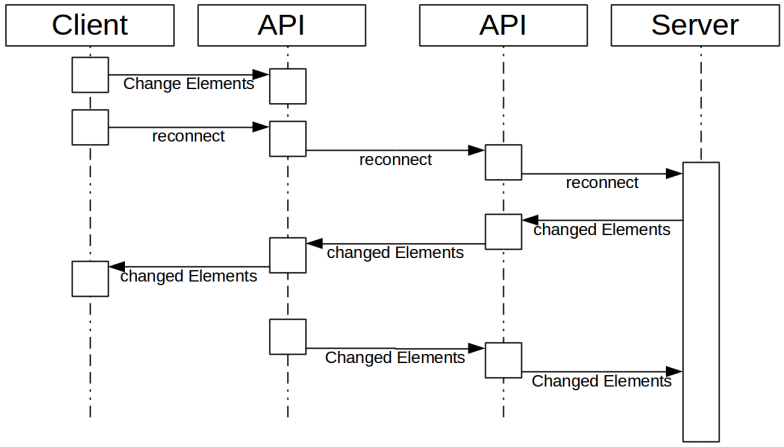


Abbildung 7: Ablauf der Synchronisation

Serverfunktion

Der Aufruf einer Serverfunktion wird asynchron durchgeführt. Der Aufruf wird zusammen mit den dazu gehörenden Parametern in Form einer

Nachricht an den Server gesendet. Die geänderten Elemente werden anschliessend vom Server, wie im Abbildung 8, retourniert.

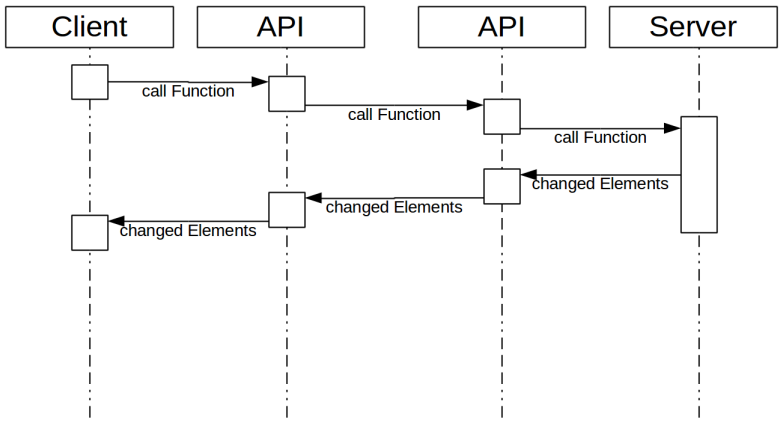


Abbildung 8: Ablauf eines Aufrufs einer Serverfunktion

7.3.7 *Flux Architektur*

Das Flux Paradigma[Fis14] ist eine Applikationsarchitektur welche sehr stark auf das Konzept der nachrichtenbasierten Kommunikation basiert und somit auch einen unidirektionalen Datenfluss, wie in Abbildung 9 vorgibt. Daten können nur über das versenden einer Nachricht manipuliert werden. Sowohl Views als auch die API können Aktionen auslösen, und so den Datenbestand mutieren.

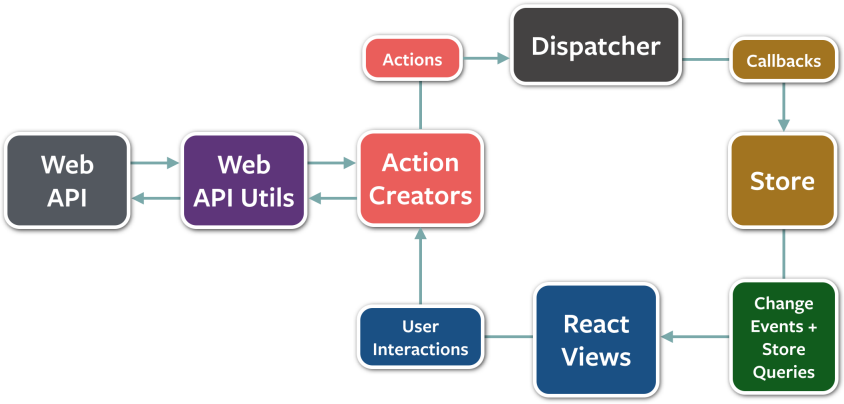


Abbildung 9: Flux Diagramm [Fis14]

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

7.4 BEISPIELAPPLIKATION

Die Aufgabenstellung verlangt, dass der Prototyp anhand eines passenden Fallbeispiels die Funktionsfähigkeit und Praxistauglichkeit der Synchronisations- und Konfliktlösungs-Verfahren zeigt. Dazu wird das Fallbeispiel "Synchronisation von Kontakten" umgesetzt. Die Umsetzung konzentriert sich auf die Synchronisationsverfahren und lässt Aspekte der Benutzerverwaltung und Usability, bewusst weg.

Die Beispielapplikation muss aufgrund der Anforderungen der Anforderungsanalyse und der Beschreibung des Fallbeispiels, auf der für den Endbenutzer sichtbaren Ebene, folgende Funktionalität aufweisen:

1. Ein neuer Kontakt kann erfassen werden.
2. Ein bestehender Kontakt kann mutiert werden.
3. Ein bestehender Kontakt kann gelöscht werden.
4. Die gesamten Funktionalität ist online wie auch offline verfügbar.

TEIL IV

IMPLEMENTIERUNG UND TESTING

8 PROTOTYP

In diesem Kapitel wird die Implementation des Prototypen gemäss den Anforderungen aus dem Kapitel Analyse adressiert, sowie auf die dafür verwendeten Technologien und Frameworks eingegangen.

8.1 UMSETZUNG KONFLIKTAUFLÖSUNGSVERFAHREN

Die Implementation der Konfliktauflösungsverfahren "Zusammenführung", "kontextbezogene Zusammenführung" sowie die traditionelle Synchronisation ist mit wenigen Zeilen Code implementierbar.
Die traditionelle Synchronisation überprüft auf Zeile 3 ob das Attribut des übermittelten Objekts (*new_obj*) auf dem Server (*db_obj*) bereits verändert wurde und übernimmt, nur falls dies nicht der Fall ist, das neue Attribut.

```
1 traditional: (data, db_obj, new_obj, prev_obj, attr) ->
2     if new_obj[attr]?
3         data[attr] = if prev_obj[attr] is db_obj[attr]
4         then new_obj[attr] else db_obj[attr]
```

Zur Umsetzung des Konzepts der Zusammenführung wird auf Zeile 3 überprüft, ob das Attribut des übermittelten Objekts in dieser Nachricht mutiert wurde und nur, falls dies der Fall ist, das neue Attribut übernommen.

```
1 combining: (data, db_obj, new_obj, prev_obj, attr) ->
2     if new_obj[attr]?
3         data[attr] = if new_obj[attr] is prev_obj[attr]
4         then db_obj[attr] else new_obj[attr]
```

Die kontextbasierte Zusammenführung wird umgesetzt indem auf Zeile 4 geprüft wird, ob das Kontextattribut des Objekts auf dem Server bereits mutiert wurde. Nur wenn dies nicht der Fall ist, wird das neue Attribut übernommen.

```
1 contextual: (data, db_obj, new_obj, prev_obj,
2                 attr, cont) ->
3     if new_obj[attr]?
4         data[attr] = if prev_obj[cont] is db_obj[cont]
5         then new_obj[attr] else db_obj[attr]
```

8.2 UMSETZUNG KONFLIKTVERHINDERUNGSVERFAHREN

Da der Prototyp durch sein Design bereits die Unterschiedsbasierte Synchronisation einsetzt und die Konfliktauflösung bereits auf Ebene der Attribute durchführt und somit das Konzept der Update Transformation bereits

unterstützt, muss nur das Konzept der Serverfunktion gesondert implementiert werden.

8.3 ENTWICKLUNG

Die beim Server über die Verbindung zum Client eingehenden Nachrichten, werden über den Nachrichtenbus weitergeleitet. Auf Zeile 3 wird die Server-Interne *dispatch*-Funktion mit dem Nachrichtennamen und der tatsächlichen Nachricht aufgerufen und somit auf den Nachrichtenbus publiziert.

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

Sowohl über die API eingehende Nachrichten, als auch interne Nachrichten des Servers werden gleichwertig behandelt. Alle Nachrichten werden gleichermassen über den Nachrichtenbus verteilt.
Auf dem Client werden eingehenden Nachrichten nur in den Nachrichtenbus des Clients eingespeist, wenn diese gültig sind. Auf Zeile 2 wird überprüft ob der Name der empfangenen Nachricht gültig ist. Falls dem so ist, wird dem clientseitigen Nachrichtenbus eine neue Nachricht übergeben. Um später zu erkennen, ob die Nachricht von einer View oder der API selbst kam, wird das Flag *updated* gesetzt.

```
1 @.io.on 'message', ( msg ) ->
2     if msg.messageName is 'C_PRES_STORE_update'
3         flux.doAction 'C_PRES_STORE_update',
4         meta:
5             model:msg.message.meta.model
6             updated:true
7         data:
8             msg.message.data
```

Die Struktur der Daten muss nur auf dem Server definiert werden. Der Client selbst übernimmt automatisch die vom Server verwendete Struktur. Lediglich der Name des Objekts muss in der Storedefinition des Clients eingetragen werden (Zeile 2 und 8). Die Datenhaltung des Clients selbst reagiert auch auf die *update* Nachrichten (Zeile 7) und aktualisiert sich dem entsprechend.

```
1 flux.createStore
2     id: "prototype_contact",
3     initialState:
4         contacts : []
5
6     actionCallbacks:
7         C_PRES_STORE_update: ( updater, msg ) ->
8             if msg.meta.model is "Contact"
9                 ...
```

Die Datenstruktur wird auf dem Server direkt in den Datendefinitionen von Sequelize eingetragen. Sequelize stellt dafür verschiedene Datentypen zur Verfügung. Die so definierten Objekte können automatisiert in einem relationalen Datenbanksystem abgespeichert werden.

```
module.exports = (sequelize, DataTypes) ->
    Contact = sequelize.define "Contact", {
        first_name: DataTypes.STRING
        last_name: DataTypes.STRING
        ...
    }, {}
```

Die Konfiguration der Konfliktauflösungsstrategie wird in der Logikschicht für jedes Attribut einzeln definiert.

```
_traditional data, db_objs, me.obj, me.prev, 'first_name'
_traditional data, db_objs, me.obj, me.prev, 'last_name'
```

8.4 TECHNOLOGIE STACK

Die für die Entwicklung eingesetzten Technologien und Frameworks sind in der Tabelle 8 aufgeführt.

Tabelle 8: 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 ist, wurde Grunt eingesetzt.
Karma	Karma ist ein Testrunner, der Tests direkt im Browser ausführt. Weiter können automatisch Coverage-Auswertungen durchgeführt werden.
CoffeeScript	CoffeeScript ist eine einfach zu schreibende Sprache die zu JavaScript kompiliert wird. 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 Kompilierzeit 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 Truth" verhindert darüber hinaus das Auftreten von Anzeigefehler.
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 Web-Framework 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.

8.5 ENTWICKLUNGSUMGEBUNG

Die Entwicklungsumgebung ist so portabel wie möglich gestaltet. Alle benötigten Abhängigkeiten sind automatisiert installierbar. Die dazu nötigen Befehle sind nachfolgend aufgeführt und müssen direkt im Repository ausgeführt werden.

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

8.6 GRAPHISCHE UMSETZUNG

Bei der Umsetzung des GUI sind die Richtlinien Material Design[ˆmatdesign] angewendet worden.
[ˆmatdesign] https://www.google.com/design/spec/material-design/introduction.html

8.6.1 *Kontaktübersicht*

Alle im System erfassten Kontakte werden beim Aufrufen der Web-Applikation dem Benutzer angezeigt. Die wichtigsten Attribute wie Name, Telefonnummer und Email-Adresse werden übersichtlich aufgelistet.

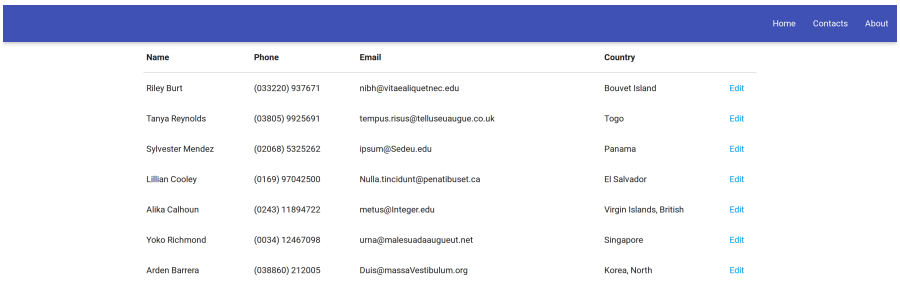


Abbildung 10: Kontaktübersicht

8.6.2 *Kontakt Detailansicht*

Die Detailansicht des Kontakts zeigt alle Attribute des Kontakts, gruppiert nach Zusammengehörigkeit, an. So sind die Attribute Nachname, Vorname, akademischer Grad sowie der Mittelname auf einer Zeile zusammengefasst. Die Adresse mit den Attributen Land, Kanton, Stadt und Strasse sind darunter auf zwei Zeilen verteilt. Die Email-Adresse sowie Telefonnummer ist zuunterst aufgeführt.

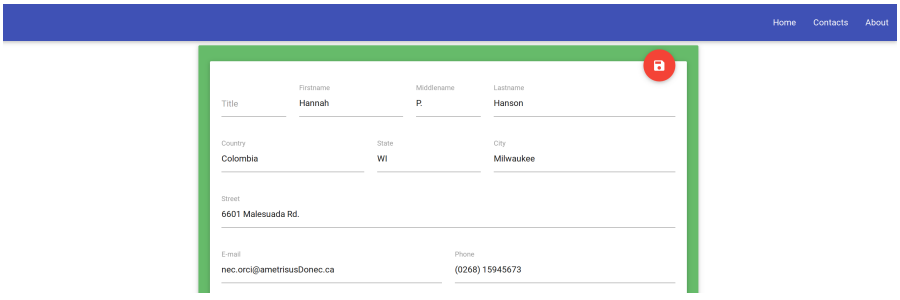


Abbildung 11: Kontakt Detailansicht

9 TESTING

Da während der Entwicklung des Prototypen viel Wert auf eine stabile Implementation gelegt wird, wir die gesamte Codebasis automatisch und manuell getestet sowie automatisiert analysiert. In den nachfolgenden Kapiteln werden die beiden Methoden zur statischen Analyse und zum Testen kurz ausgeführt.

9.1 UNIT-TESTING

Die Testrunner-Suite Karma erlaubt es Programmcode fortlaufend zu testen. Dabei werden automatisch bei einer Änderung des Codes, die gesamten Tests erneut durchgeführt und das Ergebnis ansprechend dargestellt. Die Tests selbst werden mit der Test-Suite Jasmine durchgeführt.

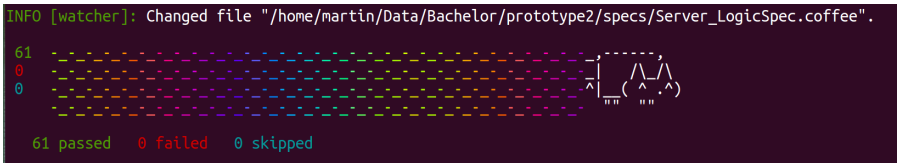


Abbildung 12: Karma Testrunner

9.2 COVERAGE ANALYSE

Karma erlaubt weiter bei jedem Durchlauf der Tests auch die Durchführung einer Coverage-Analyse. Dafür wir die Coverage-Suite Istanbul verwendet. Neben der Anzahl, in den Tests durchlaufenen Befehle und Zeilen und aufgerufenen Funktionen wird auch angezeigt wie viele Verzweigungen durchlaufen und durch Tests abgedeckt werden.

Abbildung 13: Istanbul Coverage

9.3 MANUELLES TESTEN

Die durchgeführten Tests und Analysen decken nahezu die gesamte Codebasis ab. Durch den Einsatz der nachrichtenbasierten Architektur sind die Schnittstellen zwischen den einzelnen Bausteinen und Layers bereits sehr gut mit Unit-Testing überprüfbar.
Integrationstests werden deshalb nur manuell durchgeführt, damit die vor-

handene Entwicklungsumgebung nicht durch den Einsatz dafür geeigneter Tools, wie Selenium, aufgebläht wird.
Im nachfolgenden sind die Tests basierend auf den Akzeptanzkriterien der Anforderungsanalyse an den Prototypen aufgeführt.

Tabelle 9: Manuelle Tests

MT	**AC**	**Status**	**Datum**	**Beschreibung**
T1	AC01	OK	21.05.2015	Initiale Synchronisation wurde ausgeführt.
T2	AC02	OK	10.05.2015	Mutationen können ohne Verbindung zum Server durchgeführt werden.
T3	AC03	OK	21.05.2015	Mutationen werden bei Verbindung zum Server übertragen.
T4	AC04	OK	21.05.2015	Die Konfliktauflösung wird bei der Synchronisation durchgeführt.

TEIL V

ABSCHLUSS UND AUSBLICK

10 REVIEW

10.1 LIMITATIONEN

10.2 OFFENE FRAGESTELLUNGEN

Q1: Welcher Einsatzbereich bestehen für das Multistate-Konzepts und wo liegt der Nutzten davon?
Q2: Ist es nötig auf wiederholte, gleiche Abfragen zu unterschiedlichen Zeitpunkte, die selben Resultate zu liefern?
Q3: Wo liegen die Chancen und Probleme beim Einsatz von neuronalen Netzen zur Auflösung von Konflikten?

10.3 AUSBLICK

Es konnte zwar gezeigt werden, dass eine Vereinfachung und Generalisierung des schweren Synchronisationsproblems möglich ist, ein benutzerfreundliches Framework, welches dabei unterstützt, wurde jedoch nicht erarbeitet. Gerade mit einem flexiblen und von jedem einsetzbaren Framework, würde der Fortschritt im Bereich mobiler Applikationen weitaus schneller voran schreiten.
Die Grundlagen für ein solches Framework wurden mit dieser Arbeit gelegt, eine produktiv einsetzbare Lösung bedarf jedoch noch viel Arbeitsaufwand.

10.4 VALIDATION

Die umgesetzten Konzepte haben gut funktioniert. In Nachfolgenden wird die Erfüllung der Aufgabenstellung und die Erstellung der Deliveries überprüft. Dafür wird die Fragestellung aus dem Kapitel Detailanalyse der Aufgabenstellung erneut durchlaufen und darauf geprüft.

Resultat	Delivery	Beschreibung	Status
R1	D1,D2	Wurden die technischen Grundlagen dokumentiert?	erfüllt
R2	D3, D4	Wurden die bestehenden Synchronisations- und Replikationsverfahren analysiert?	erfüllt
R3	D5, D6	Wurden Konzepte zur Synchronisation sowie zur Implementation eines Prototypen erarbeitet?	erfüllt
R4	D7	Wurde ein Prototyp mit den ausgewählten Verfahren implementiert?	erfüllt

R5	D8	Wurde die Funktionsfähigkeit des Prototypen getestet?	erfüllt

11

DAS WORT ZUM SCHLUSS

Im Verlaufe dieser Bachelorarbeit durfte ich mich intensiv mit Web-Technologien und Synchronisationsverfahren auseinandersetzen und so einen tiefen Einblick in die Techniken und Standards gewinnen. Neben dem persönlichen Interesse daran konnten Richtlinien erarbeitet werden, die das sehr schwere Problem der Synchronisation von Daten aufbricht.
Ich konnte im Verlauf des Projekts alle gesetzten Ziele erreichen. Eine Analyse der möglichen Synchronisationsverfahren und eine Analyse auf deren Umsetzbarkeit hin, wurde durchgeführt und die ausgewählten Konzepte umgesetzt.
Die breite Abstützung des gewählten Themas, sowie die Terminplanung dieser Arbeit war durchaus gewagt, gerade aber diese Freiheit bezüglich des Themas erlaubte es mir, tief in das Themengebiet vorzudringen. Der straffe Zeitplan verhinderte dabei zu stark abzudriften.
Ich habe mich bei der Anforderungsanalyse an das Vorgehensmuster aus dem Buch *"Basiswissen Requirements Engineering"* gehalten. Rückblickend ist für die Implementation eines Prototypen, anhand welchem die Umsetzbarkeit eines Konzepts gezeigt werden soll, eine solch aufwändige Planung zu zeitintensiv. Eine Beschränkung auf die Erhebung der Use-Cases wäre durchaus genügend gewesen.
Ich bin mit den Resultaten meiner Arbeit sehr zufrieden. Nicht nur konnte ich selbst viel neues Lernen sondern auch erfahren dass gewisse Lösungsansätze in der Praxis nicht problemlos umsetzbar sind. Es hat mir viel Spass bereitet neue Technologien und Lösungsansätze zu erkunden.

TEIL VI

ANHANG

A

GLOSSAR

ORM
ORM steht für object-relational mapping und ist eine Technik mit der Objekte einer Anwendung in einem relationalen Datenbanksystem abgelegt werden kann.

Node
Node oder Node.js ist eine Plattform welche es erlaubt JavaScript serverseitig auszuführen.
https://nodejs.org/

RPC
Remote Procedure Call ist eine Technologie um Funktionsbausteine in einem anderen Prozess aufzurufen.

Jasmine
Jasmine ist ein verhaltensbasiertes Testframework für JavaScript.
http://jasmine.github.io

Karma
Karma ist ein Testrunner-Framework zur kontinuierlichen Ausführung von UnitTests.
http://karma-runner.github.io

Mocks
Mocks sind Code-Attrappen, die es ermöglichen, noch nicht vorhandene oder nicht verfügbare, Funktionalitäten und Objekte zu simulieren.
http://de.wikipedia.org/wiki/Mock-Objekt

Github
Github ist ein Cloud basierter SourceCode Verwaltungsdienst für Git.
https://github.com

B

AUFGABENSTELLUNG

B.0.1 *Thema*

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

B.0.2 *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?

B.0.3 *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.

B.0.4 *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

B.0.5 *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

B.1 DETAILANALYSE DER AUFGABENSTELLUNG

Die Detailanalyse der Aufgabenstellung formuliert die zu erbringenden Aufgaben aus und unterteilt diese in Deliveries *D1* bis *D8*.

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 (*D1*), sowie eine Zusammenfassung (*D2*) 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 aktuellen Systeme an ihre Grenzen stossen.
Weiter muss eine Anforderungsanalyse für eine Beispielapplikation durchgeführt werden.
Erwartet wird sowohl die Dokumentation der Synchronisationsverfahren (*D3*) als auch das Ergebnis der Anforderungsanalyse (*D4*).

Konzept

Die Erarbeitung und Überprüfung der Umsetzbarkeit neuer Synchronisationskonzepte wird in der Konzeptionsphase gefordert.
Erwartet wird eine Darstellung der erarbeiteten Konzepte (*D5*) und eine Konzeption zur Umsetzung (*D6*) derer 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 (*D7*) implementiert.

Review

Das Review soll eine Retrospektive auf die erarbeiteten Resultate werfen und kritisch hinterfragen.
Erwartet wird ein Test des Prototyps sowie ein Protokoll der durchgeführten Tests (*D8*).

C

PROJEKTMANAGEMENT

C.1 PROJEKTPLANUNG

Der Projektplan (14) illustriert die Strukturierung des Projekts über die gut 6 Monate lange Projektzeit.

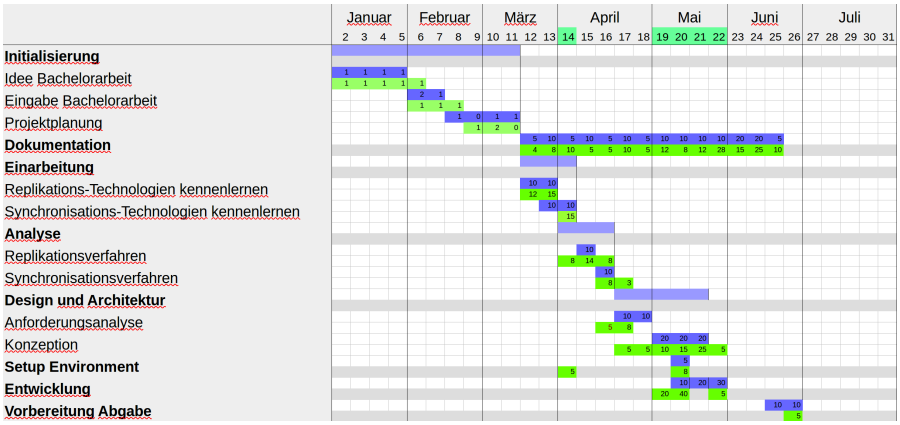


Abbildung 14: Projektplan

C.1.1 *Aufwandschätzung*

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

Arbeitsschritt	Aufwand in Stunden
Initialisierung	10
Recherche (*D2*,*D3*)	45
Analyse (*D2*,*D3*,*D4*)	20
Konzeption (*D5*,*D6*)	80
Prototyp (*D7*,*D8*)	60
Dokumentation (*D1*)	135
Abgabe	20
Total	**370**

C.2 RAHMENBEDINGUNGEN

Der offizielle Projektstart ist der 18. März 2015. Das Projekt muss bis spätestens 11.08.2015 abgegeben werden.
Während der Kalenderwochen 14, 19, 20, 21, und 22 hat der Student Urlaub und kann deshalb während dieser Zeit intensiv der Bearbeitung der Thesis widmen.

Termin	Datum	Bemerkungen
Kick-Off	18.03.2015	-
Design Review	20.05.2015	Der Entscheid über das Abgabedatum muss am 06.06.2015 gefällt werden.
Abgabe-Entscheid	06.06.2015	Die Thesis wird am 30.06.2015 abgegeben.
Abgabe Bachelorthesis	30.06.2015	-
Abschlusspräsentation	02.06.2015	-

C.3 SOLL/IST ANALYSE

Arbeitsschritt	Soll	Ist
Initialisierung	10	5
Recherche	45	47
Analyse	20	41
Konzeption	80	78
Prototyp	60	65
Dokumentation	135	157
Abgabe	20	5
Total	**370**	**412**

Der Mehraufwand von ~10% beruht vornehmlich darauf, dass das Sammeln und Verstehen von Informationen über bestehende Synchronisationsverfahren (Analyse) und das Verfassen der Arbeit selbst, sich als deutlich zeitintensiver als geplant herausgestellt hat.

C.4 DOKUMENTATION

Da die Nachvollziehbarkeit von Änderungen in MS Word sehr umständlich ist, habe ich in Betracht gezogen, die Arbeit mit 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 auch die Latex-Syntax.

C.5 VERSIONSVERWALTUNG

Damit einerseits die Daten gesichert und andererseits die Nachvollziehbarkeit von Änderungen gewährleistet ist, verwende ich git. Das Repository [1] ist für den Betreuer und Experten jederzeit einsehbar.

1 https://github.com/eigenmannmartin/Bachelor

D

ANFORDERUNGSANALYSE

D.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.
Die Schlüsselwörter „muss“, „muss nicht“, „erforderlich“, „empfohlen“, „sollte“, „sollte nicht“, „kann“ und „optional“ in allen folgenden Abschnitten sind gemäss RFC 2119 zu interpretieren. [Bra97]

D.2 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.
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.

UseCase	
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	-

D.3 ANFORDERUNGEN

In diesem Kapitel sind alle funktionalen und nicht-funktionalen Anforderungen aufgeführt die aus den UseCases resultieren. Der entsprechende UseCase ist dabei jeweils referenziert.

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.

FREQ04.01 Löschen eines Objekts

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

FREQ04.02 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.

D.4 AKZEPTANZKRITERIEN

In den nachfolgenden Tabellen findet sich eine Aufführung der Akzeptanzkriterien, basierend auf den bereits erarbeitete Anforderungen.

AC01 Initiale Synchronisation

Akzeptanzkriterium	
REQ-Referenz	FREQ01.01, FREQ01.02, FREQ04.02
Vorbedingung	Der Client hat eine Verbindung zum Server aufgebaut.

Akzeptanzkriterium	
Kriterium	Beim Starten des Clients wird der gesamte Datenbestand des Servers an den Client übermittelt.

AC02 Lokale Mutationen

Akzeptanzkriterium	
REQ-Referenz	FREQ04.01, FREQ04.02
Vorbedingung	Der Client hat bereits eine initiale Synchronisation durchgeführt.
Kriterium	Jedes Element des lokalen Datenbestand des Clients kann gelesen, mutiert und gelöscht werden. Neue Elementen können dem Datenbestand hinzugefügt werden.

AC03 Synchronisation

Akzeptanzkriterium	
REQ-Referenz	FREQ02.01, FREQ02.02, FREQ03.01, FREQ05.01, FREQ05.02, NFREQ01, NFREQ02
Vorbedingung	Der lokale Datenbestand des Clients wurde mutiert und noch nicht synchronisiert. Der Client hat eine Verbindung zum Server aufgebaut.
Kriterium	Jede auf dem Client durchgeführte Mutation wurde aufgezeichnet und wird dem Server in der aufgezeichneten Reihenfolge übermittelt.

AC04 Konfliktbehandlung

Akzeptanzkriterium	
REQ-Referenz	NFREQ02
Vorbedingung	Der Client hat eine Verbindung zum Server aufgebaut. Eine Synchronisation wurde durchgeführt.
Kriterium	Das Ergebnis der Konfliktauflösung wird dem Client übermittelt.

E

VERZEICHNISSE

E.1 QUELLENVERZEICHNIS

[Bra97] S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. https://www.ietf.org/rfc/rfc2119.txt. [Online, accessed 22-Mai-2015]. 1997.

[CB10] 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.

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

[Elg14] 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.

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

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

[Fie00] 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.

[Fis14] 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.

[Lip14] 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.

[Mei10] Andreas Meier. *Relationale und postrelationale Datenbanken*. Bd. 7. eXamen.pressSpringerLink. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. ISBN: 978-3-642-05256-9.

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

[Nak09] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. https://bitcoin.org/bitcoin.pdf. 2009.

E.2 TABELLENVERZEICHNIS

E.3 ABBILDUNGSVERZEICHNIS

F

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.

G

PERSONALIENBLATT

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

H

BESTÄTIGUNG

Hiermit bestätigt der Unterzeichnende, dass die Bachelorarbeit mit dem Thema "Evaluation von Synchronisations- und Konfliktlösungsverfahren im Web-Umfeld" gemäss freigegebener Aufgabenstellung mit Freigabe vom 09.02.2015 ohne jede fremde Hilfe im Rahmen der gültigen Reglements selbständig ausgeführt wurde.
Alle öffentlichen Quellen sind als solche kenntlich gemacht. Die vorliegende Arbeit ist in dieser oder anderer Form zuvor nicht zur Begutachtung vorgelegt worden.

St.Gallen den 30.06.2015

Martin Eigenmann