Evaluation von Synchronisations- und Konfliktlösungsverfahren im Web-Umfeld

Martin Eigenmann

29. Mai 2015

1

Abstract

2

Danksagungen

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

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

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

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

3

Inhaltsverzeichnis

4

Personalienblatt

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

5
Bestätigung

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

St.Gallen 1.05.2015

Martin Eigenmann

Teil I.

Einleitung und Abgrenzung

6
Einleitung

6.1. Motivation und Fragestellung

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

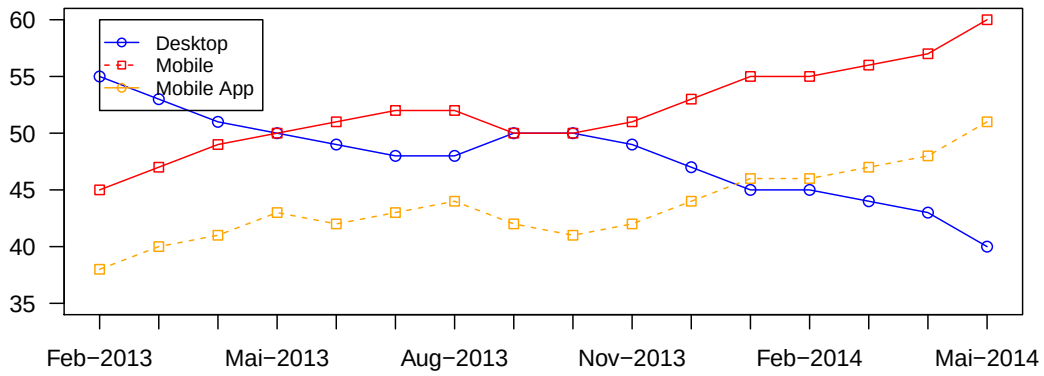


Abbildung 6.1.: Verteilung der online verbrachten Zeit nach Platform (Grafik erstellt gemäss der Daten von [6])

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

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

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

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

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

Mein starkes persönliches Interesse zur Analyse und Verbesserung dieses Systems treibt mich an, diese Arbeit zu starten.

6.2. Aufgabenstellung

Die von der Leitung des Studiengangs Informatik freigegebene Aufgabenstellung ist im Appendix unter „Aufgabenstellung" aufgeführt.

6.3. Abgrenzung der Arbeit

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

Ich möchte in dieser Arbeit einen allgemeinen Ansatz erarbeiten und die Grenzen dessen ausloten, um so zu zeigen, wo die Grenzen 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 wurde drei Anwendungsfälle erarbeitet, auf welchen die Untersuchungen durchgeführt wurde.

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

6.4. Sprache

6.5. Richtlinien

7
Aufbau der Arbeit

Teil i - Einleitung und Abgrenzung

Teil ii - Technische Grundlagen und Architekturen

Erarbeitung bekannter Verfahren, etc.

Konzeption - Konzept

Analyse der Daten
Erarbeitung des Konzepts

Implementation - Implementierung und Testing

Erstellen des Prototypen etc.

Teil iv - Abschluss und Ausblick

Teil II.

Technische Grundlagen und Architekturen

8
Recherche

Dieses Kapitel erklärt die wichtigsten Grundbegriffe und wiedergibt die während der Recherche gesammelten Informationen.

8.1. Fachbegriffe

Eine Aufführung und Erlährung der Fachbegriffe befindet sinch im Appendix unter „Glossar“

8.2. Erläuterung der Grundlagen

8.2.1. Datenbanken

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

- Atomarität
- Konsistenzerhaltung
- Isolation
- Dauerhaftigkeit

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

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

8.2.2. Monolithische Systeme

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

8.2.3. Verteilte Systeme

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

physisch verteilte Systeme

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

logisch verteilte Systeme

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

8.2.4. Verteilte Algorithmen

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

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

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

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

8.2.5. Verteilte Datenbanken

[Präsenzbibliothek ZHAW]

8.2.6. Replikation

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

Synchrone Replikation

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

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

Asynchrone Replikation

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

Merge Replikation

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

[6] So kann der Backup-Knoten nur Nachts über verfügbar sein, damit die dazwischen liegende Verbindung 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.

8.2.7. Block-Chain

Die Block-Chain ist eine verteilte Datenbank die ohne Zentrale Kontrolle 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. [9]

8.3. Replikationsverfahren

8.3.1. MySQL

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

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

Der Master-Master Betrieb erlaubt die Mutation des Datenbestand auf allen Replikationsteilnehmern. Dies wird mit dem 2-phase-commit (2PC) Protokoll erreicht.

MySQL schliesst Konflikte beim Betrieb eines Master-Master Systems, durch die Implementation des 2PC-Protokoll aus.

2-Phase-Commit-Protocoll

Um eine globale 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.

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

Bei einer mutierenden Operation auf dem Datenbestand, müssen sich alle beteiligten Datenbanksysteme in einem operationalen Zustand befinden und inder Lage sein, miteinander zu kommunizieren. [8]

[9] Das ändern vorhergehender Einträge benötigt mehr Rechenzeit, als alle anderen Teilnehmer ab diesem Zeitpunkt 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 Ausführung.

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

8.3.2. MongoDB

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

[Bi-Directional Replica-Set (rick446/mmm)?]

8.4. Synchronisationsverfahren

8.4.1. Backbone.js

Das Web-Framework Backbone.js implementiert zur Datenhaltung und Synchronisation derer, einen REST kompatiblen Key-Value Store. Der Key-Value Store kann entweder ein einzelnes Element oder ein gesamtes Set an Elementen[12] von der REST-API lesen und ein einzelnes Element schreibend an die REST-API übermitteln. Die der REST-API zugrundeliegende Datenquelle kann beliebig gewählt werden.

Der REST-Standard verlangt dass für ein Element immer der gesamte Status übermittelt wird. So wird bei lesendem und schreibenden Zugriff immer die gesamte Repräsentation eines Elements gesendet wird, also nicht von einem Kontext ausgegangen werden kann. [4]

Backbone.js sieht keinen Mechanismus vor, der Änderungen auf dem Server automatisch auf den Client übernimmt. Es muss entweder periodisch die gesamte Collection gelesen oder periodisch auf ein Änderungs-Log zugegriffen werden, um Änderungen einer Collection zu erkennen.

Darüber hinaus entscheidet der Server alleine, ob eine konkurrierende Version des Clients übernommen wird. Backbone.js sendet beim Synchronisieren eines Elements, dessen gesamten Inhalt an den Server und übernimmt anschliessend die von der REST-API zurückgelieferten Version. Die zurückgelieferte Version, kann eine bereits auf dem Server als aktiv gesetzt Version des Elements sein. Der Server kann also frei entscheiden welche Version des Elements aktiv wird. Aktualisierung die nicht berücksichtige werden, gehen üblicherweise verloren[13].

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

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

[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 des Servers.

8.4.2. Meteor.js

Das WEB-Framework Meteor.js implementiert eine Mongo-Light-DB im Client-Teil und synchronisiert diese über das Distributed Data Protocol 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 Element 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 eine Änderung auf dem Client erfolgreich an den Server übermittelt und übernommen wird. Da Mutationen üblicherweise zeitnah übertragen werden, treten nur in seltenen Fällen Konflikte auf.

1. Im ersten Schritt wird ein neues oder die mutierten Attribute eines Elements 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 Element entsprechen.

9 Analyse

Dieses Kapitel setzt sich mit der Klassifikation und Typisierung von Daten auseinander. Es wird eine Klassifikation erarbeitet und anhand von Beispielen gezeigt, dass diese auch anwendbar ist.

9.1. Synchronisationsproblem

Die Überprüfung der Klassifikation wird anhand der folgenden zwei Problemstellungen durchgeführt.
Beide Problemstellungen sind so gewählt, dass sie zusammen einen möglichst allgemeinen Fall abdecken und so die Überprüfung aussagekräftig bleibt.

9.1.1. Synchronisation von Kontakten

In vielen Anwendungsszenarien müssen Kontaktdaten abgeglichen werden. Dieses Beispiel beleuchtet die Synchronisation einer Firmen-Kontaktdatenbank mit mobilen Clients der Mitarbeiter.
Verkaufsmitarbeiter müssen jederzeit Kontaktdaten abfragen, neu erfassen und anpassen können, ohne dafür mit dem zentralen Server verbunden zu sein. Gerade in wenig entwickelten oder repressiven Ländern ist eine ständige Verbindung nicht immer gegeben.

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

Tabelle 9.1.: Attribute Firmen-Kontakt

Attribut	**Beschreibung**
Name	Der gesamte Name (Vor-, Nach,- und Mittelname) der (Kontakt-)Person.
Adresse	Die vollständige Postadresse der Firma oder Person.

Attribut	Beschreibung
Email	Alle aktiven und inaktiven Email-Adressen des Kontakts. Jeweils nur eine Email ist die primäre Email-Adresse.
Telefon	Alle aktiven und inaktiven Telefonnummern des Kontakts. Jeweils nur eine Telefonnummer ist die primäre Nummer.
pNotes	Persönliche Bemerkungen zum Kontakt. Nur der Autor einer Notiz, kann diese bearbeiten oder lesen.

Im Folgenden sind vier typische Szenarien beschrieben.

1. Szenario
Erfassen einer neuen Telefonnummer zu einem bestehenden Kontakt.

2. Szenario
Ändern der primären Telefonnummer eines bestehenden Kontakts.

3. Szenario
Ändern des Namens einer Kontaktperson eines bestehenden Kontakts.

4. Szenario
Hinzufügen/Ändern der (persönlichen) Zusatzinformationen eines bestehenden Kontakts.

9.1.2. Syncrhonisation eines Service Desks

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

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

Tabelle 9.2.: Attribute Support-Fall

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

Die vier typischsten Aufgabenszenarien sind hier aufgeführt.

1. Szenario
Erfassen eines neuen Support-Falls.

2. Szenario
Einem Support-Fall eine neuen Anmerkung hinzufügen.

3. Szenario
Erfassen von Arbeitszeit auf ein Ticket.

4. Szenario
Zu einem Supportfall eine Lösung (FAQ-Eintrag) erfassen.

9.2. Datenanalyse

Um eine klares Verständnis der Daten zu erhalten, wird analysiert, bezüglich welcher Eigenschaften, Daten unterschieden werden können.

Daten können bezüglich ihrer Beschaffenheit, Geltungsbereich und Gültigkeitsdauer unterschieden werden. Dabei spricht man von der Klassifikation.
Die Datentypisierung hingegen, unterscheidet nach dem äusseren Erscheinungsbild der Daten. Zu beachten gilt dass eine Attributsgruppe, also eine logische Aufteilung der Informationen in mehrere einzelne Attribute, eine zusammenhängende Informationseinheit ist.

Zur Klassifikation werden nur die in den Daten enthaltenen Informationen herangezogen. Die Form der Daten, also der Datentyp selbst ist für die Klassifikation unerheblich.
Weiter kann die Klassifikation zwischen Struktur und Art der Daten unterscheiden.

9.2.1. Klassifikation nach Art

Die Art der Daten wird entweder explizit durch die Art der Implementation (temp, static, dynamic) oder implizit durch die Zugriffsstruktur (excl, publ) definiert. Es ist keine Kenntnis über die Daten nötig, eine Statische Analyse der Zugriffe auf die Daten reicht aus, um die Daten-Art zu bestimmen.

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

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

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

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

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

9.2.2. Klassifikation nach Struktur

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

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

Unabhängige Daten sind selbst beinhaltend und gewinnen durch andere Daten selbst nicht mehr an Informationsgehalt.

9.2.3. Datentypisierung

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

9.3. Datenanalyse der Synchronisationsprobleme

Nachfolgend sind die Attribute eines Kontakts aus dem ersten Beispiel „Synchronisation von Kontakten“ sowie „Synchronisation eines Service Desks“ entsprechend der erarbeiteten Klassifikation und Typisierung aufgeführt.

Synchronisation von Kontakten
Die Attribute Adresse, Email und Telefon sind abhängig vom Namensattribut. Dies kommt deshalb, weil der Name der primäre Identifikator ist. Adresse, Email und Telefon sind also Kontextuell abhängig vom Identifikator.
Das Attribut pNotes hingegen ist völlig unabhängig, da es nur vom Verfasser gelesen und geschrieben werden kann und liegt daher in der Verantwortung des Dessen.

Tabelle 9.3.: Klassifikation Attribute Kontakt

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

Synchronisation eines Service Desks
Die beiden Attribute Titel und Beschreibung können nur beim Erfassen eines Support-Falls gesetzt werden. Danach bilden sie zusammen den Identifikator.
Anmerkungen können von allen Mitarbeitern erfasst und geändert werden.
Die Totale Arbeitszeit (tArbeitszeit) wird vom System errechnet und kann nicht geändert werden.

Tabelle 9.4.: Klassifikation Attribute Kontakt

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

9.4. Datenanalyse von „echten“ Fällen

Facebook

Benutzer Daten

Tabelle 9.5.: Klassifikation Attribute Facebook Benutzerdaten

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

Post

Tabelle 9.6.: Klassifikation Attribute Facebook Post

Attribut	Struktur	Art	Typ
Text	Unabhängig	exklusiv	textuell
Bild	Abhängig (Text)	exklusiv	binär
Personen	Abhängig (Bild)	exklusiv	binär
Aktion/Gefühl	Abhängig (Text)	exklusiv	textuell
Standort	Abhängig (Text)	exklusiv	binär
Likes	Abhängig (Text)	dynamisch	numerisch

Kommentar

Tabelle 9.7.: Klassifikation Attribute Facebook Kommentar

Attribut	Struktur	Art	Typ
Text	Abhängig (Post)	exklusiv	textuell
Bild	Abhängig (Text)	exklusiv	textuell

(Ob ein Kommentar angezeigt wird, entscheidet der Autor des dazugehörenden Posts)

Alle Daten werden nur von einer Person bearbeitet. -> Synchronisationsprobleme können

entstehen, jede Person ist selbst verantwortlich dass Änderungen nicht überschrieben werden. (real newest Entry wins)

Google Kalender

Termin

Tabelle 9.8.: Klassifikation Attribute Google Kalendereintrag

Attribut	Struktur	Art	Typ
Titel	Unabhängig	exklusiv	textuell
Datum	Unabhängig	exklusiv	binär
Zeit	Unabhängig	exklusiv	binär
Gäste	Unabhängig	exklusiv	binär

Es können mehrere Termine zur selben Zeit stattfinden. Der Benutzer wird dann entsprechend gewarnt. -> Synchronisationsprobleme können entstehen, jede Person ist selbst verantwortlich dass Änderungen nicht überschrieben werden. (real newest Entry wins)

9.5. Überprüfung der Klassifikation

Beide Klassen, unabhängig und abhängig werden verwendet

Es wird hauptsächlich der Typ exklusiv, und gemeinsam verwendet

==> Der Typ ist meist irrelevant und nur für die Konfliktauflösung Relevant. Dies ist aber so wie so Datenspezifisch und im Einfelfall zu begutachten.

Sinnvoll sind also noch:

Struktur: unabhängig und abhängig
Art: exklusiv, gemeinsam und dynamisch. (statisch (wil sie trotzdem geändert werden können => gemeinsam) und temp. (weil persönlich => exklusiv) sind irrelevant)

Teil III.

Konzeption

10
Konzeptansätze

Synchronisationen können so gestaltet werden, dass keine Synchronisationskonflikte auftreten, oder es können auftretende Konflikte gelöst werden.

10.1. Synchronisation (allgemein)

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

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

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

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

10.2. Datenhaltung

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

10.2.1. Singlestate

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

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

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

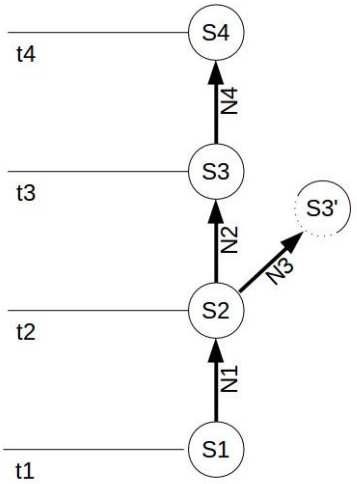


Abbildung 10.1.: Singlestate

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

10.2.2. Multistate

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

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

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

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

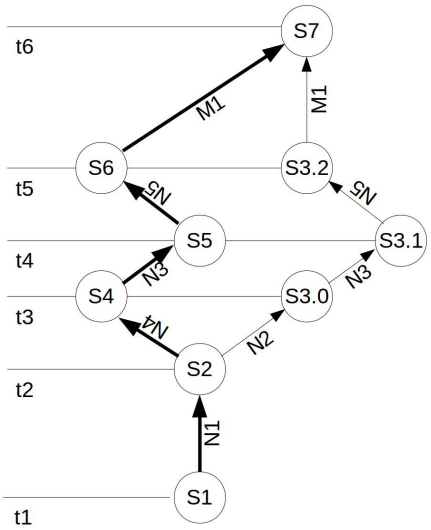


Abbildung 10.2.: Multistate

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

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

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

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

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

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

10.3. Konfliktvermeidung

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

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

10.3.1. Update Transformation

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

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

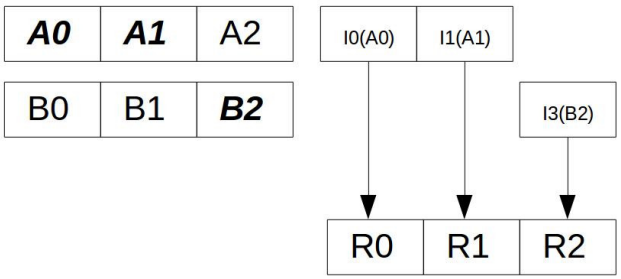


Abbildung 10.3.: Update Transformation

10.3.2. Wiederholbare Transaktion

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

10.4. Konfliktauflösung

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

10.4.1. Zusammenführung (Merge)

Einzelne Attribute oder Attributsgruppen innerhalb eines Objekts werden als eigenständige Objekte betrachtet. So kann ein Konflikt, der auftritt wenn zwei Objekte mit Mutationen in unterschiedlichen Attributsgruppen synchronisiert werden , aufgelöst werden, indem nur die jeweils mutierten Attributgrupen als synchronisationswürdig betrachtet werden.

Kontextbezogene Attribute sind in der selben Attributgruppe wie der Kontext.

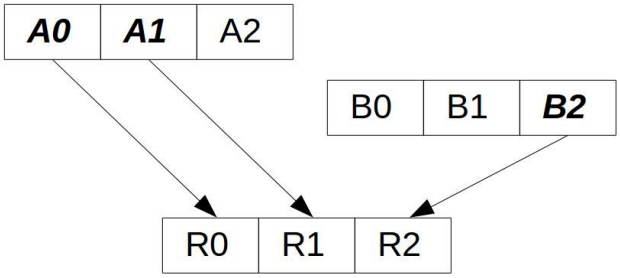


Abbildung 10.4.: Merge

Diese Operation ist unabhängig von Datentyp, Struktur und Art.
Der (automatisierte) Merge kann nur durchgeführt werden, wenn nur eine einzige Version eines Attributs existiert. -> nie konkurrierende Versionen entstehen.

Darüber hinaus versieht der Server jede Attribut-Version mit einer Versions-Nummer. So kann verhindert werden, dass ein Attribut mit einer niederen Versions-Nummer über ein neueres Attribut synchronisiert wird.

10.4.2. normalisierte Zusammenführung (Normalized Merge)

Wenn bei einer Synchronisierung mit zwei Objekten die selben Attribute mutiert wurden, kann im Falle von numerischen Attributen, das Objekt mit den geringsten Abweichungen vom Meridian über alle verfügbaren Datensätze verwendet werden. Es wird also das normalisierteste Attribut verwendet.
Bei Attributsgruppen wird die Gruppe mit der insgesamt geringsten Abweichung verwendet.
Es muss eine Abstandsfunktion für jedes Attribut oder jede Attributgruppe erstellt werden.

Es kann so die warscheinlichste Version verwendet werden. Wenn zu t_1 A_1 und zu t_2 A_2, also das selbe Attribut synchronisiert wird, wird es beide male angenommen. Beide Versionen basieren auf der gleichen Ursprungsversion. A_2 besitzt aber die kleinere Abweichung und gewinnt deshalb.
Hätte es die grössere Abweichung, würde es nicht synchronisiert werden.

10.4.3. manuelle Zusammenführung

Wenn keine Auflösung des Konflikts möglich ist, muss dieser manuell aufgelöst werden.

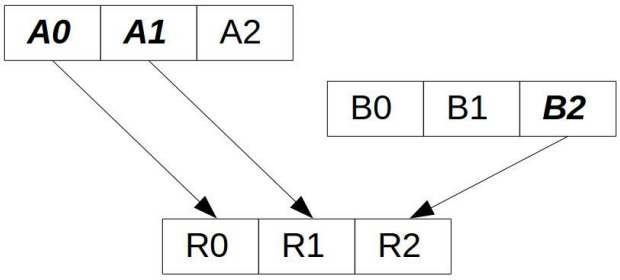


Abbildung 10.5.: Merge

10.4.4. Vergabelungs-Funktion

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

Wichtigste Information

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

Maximale Information

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

Geringste Kindsbäume

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

Online-First

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

Echte Kausalität

Der Zeitstempel der Generierung wird der Nachricht beigefügt. Die Nachricht, welche echt zu erst erstellt wurde, markiert den aktiven Teilbaum.

Timeboxing

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

11 Konzept Untersuchung

In diesem Kapitel werden die Konzeptansätze konkretisiert und auf ihre Anwendbarkeit hin untersucht.

11.1. Datenhaltung

Die trivialste Form der Datenhaltung ist eine Messagequeue. Der aktuelle Staus ist die Anwendung aller Nachrichten auf einen initialen Status erreichbar.
Der wahlfreie Zugriff auf den Staus zu jedem beliebigen Zeitpunkt zeichnet dieses einfache Design aus.

Nachfolgend findet sich eine genauere Implementations-Analyse bezüglich Single- und Multistate.

11.1.1. Singlestate

Um eine Singlestate Datenhaltung zu implementieren, ist zwingend eine Message-queue, sowie ein initialer Zustand nötig. Zur effizienteren Implementierung wird darüber hinaus ein Cache-Status benötigt, der den letzten gültigen Status enthält.

Beim Eingehen einer neuen Nachricht wird die Funktion reciveMessage() aufgerufen.

```
/* Verarbeitet eine eingehende Nachricht
 *
 * @Message.Sate         referenzierter Status
 * @Message.Mutation     Mutationsfunktion
 * @return               -
 */

reciveMessage (Message): ->

    if Message.State is @State
```

```
        @applyMsg Message
    else
        if @resolveConflict Message
            @State.apply Message.Mutation
            @MessageQueue.insert Message
        else
            break

resolveConflict(Message): ->

    if !@State.conflictsWith Message or @State.canResolvConfict Message
        return true
    else
        return false
```

Die Funktionen canResolvConfict sowie resolveConflict greifen auf den Referenzstatus der eingehenden Nachricht zu. Wie die Funktion resolveConflict genau funktioniert, wird später genauer untersucht.

Performance

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

Verbesserung

Da jede schreibende Operation zuerst den referenzierten Staus auslesen muss, und dies sehr Rechenintensiv ist, wird jeder errechneter Zustand zwischengespeichert. So ist existiert für jede Nachricht bereits ein gecachter Status. Somit werden Operationen beschleunigt und die Laufzeitkomplexität auf $O(1)$ gesenkt.

Pro/Contra

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

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

11.1.2. Multistate

Zur Implementation einer Multistate-Persistenz ist zwingend eine Messagequeue, ein Initialstatus sowie eine Shadow-Queue nötig. Zur effizienteren Implementierung wird auch hier der aktuell gültige Staus gecached.

In der Shadow-Queue werden die eingehenden Nachrichten entsprechend der referenzierten Stati geordnet und nicht mehr nach Eingangsreihenfolge.
Der Zustandsbaum wird dann entlang der Shadow-Queue aufgebaut.

Beim Eingehen einer neuen Nachricht sind folgende Schritte zu befolgen.

1. Nachricht in die Message- und Shadowqueue einsortieren
2. Zustandsbaum erneut aufbauen
3. Konflikte Auflösen

```
/* Verarbeitet eine eingehende Nachricht
 *
 * @Message.Sate          referenzierter Status
 * @Message.Mutation      Mutationsfunktion
 * @return                -
 */

reciveMessage (Message): ->

    @MessageQueue.insert Message
    @StateTree = new Tree

    for Message in @MessageQueue
        @StateTree.get(Message.State).apply Message.Mutation

    for State in @stateTree
        State.tryToResolvConflict
```

Performance

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

Verbesserung 1

Falls eine Nachricht auf einen aktuellen Zustand referenziert, muss der Baum nicht erneut aufgebaut werden.

Verbesserung 2

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

Pro/Contra

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

11.2. Konfliktvermeidung

11.2.1. 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.

```
/* Zusammenstellung einer Nachricht
 *
 * @old                 alter Status
 * @new                 neuer Status
 * @return              Message
 */

composeMessage (old, new): ->

    for AttrName, Attribut in new
        if !Attribut is old[AttrName]
            Message.Mutation[AttrName] = Attribut

    Message.State = old

    return Message
```

Es wird der gesamte „alte“ Status aber nur die geänderte Attribute des neuen Status übermittelt.

Pro/Contra

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

11.2.2. Wiederholbare Transaktion

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

Probleme/Lösungen

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

11.3. Konfliktauflösung

11.3.1. Zusammenführung

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

```
/* Zusammenstellung einer Nachricht
 *
 * @valid               aktueller Status
 * @ref                 referenzierter Status
 * @update              mutierter Status
 * @return              bool
 */

resolvConflict (valid, ref, update): ->

    NewState = new State

    for AttrName, Attribut in update
        if ref[AttrName] is valid[AttrName]
            NewState[AttrName] = Attribut
        else
```

```
            break

    return NewState
```

Probleme/Lösungen

Lösungen:
- Granularere Aufteilung des Problems

Probleme:
- keine

11.3.2. normalisierte Zusammenführung

Um eine normalisierte Zusammenführung um zu setzten, ist zwingend ein wahlfreier Zugriff auf jeden Status, sowie jede Mutationsfunktion notwendig.

Ein Konflikt bei einem Attribut wird in folgenden Schritten aufgelöst:

1. Resultate aller auf den Status angewendeten Mutationen berechnen
2. Resultat, welches am nächsten bei der Durchschnittsfunktion liegt, anwenden

```
/* Zusammenstellung einer Nachricht
 *
 * @valid               aktueller Status
 * @ref                 referenzierter Status
 * @updates             alle angewendeten Nachrichten
 * @average             durchschnittlicher Wert des Attributs
 * @return              bool
 */

resolvConflict (valid, ref, updates, average): ->

    NewState = new State
    Distances = new DistanceArray( average )

    for update in updates
        Distances.add update

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

    return NewState
```

Probleme/Lösungen

Lösungen:
- Sofern Daten dafür ausreichen / guter Kompormiss

Probleme:
- eventuell „falsche Daten“

11.3.3. manuelle Zusammenführung

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

Probleme/Lösungen

Lösungen:
- sicher korrekte Daten

Probleme:
- manueller Aufwand nötig

11.4. Zusammenfassung

12
Design des Prototypen

In diesem Kapitel werden die aus dem Kapitel [Konzept] gewonnenen Erkenntnisse so umgesetzt dass sie den Anforderungen aus dem Kapitel Anforderungsanalyse genügen.

12.1. Design-Ansätze

!!!! Verwendung von Singlestate
!!!! Anforderungsanalyse

12.1.1. Nachrichtenbasierte Architektur

Versenden einzelner Nachrichten -> Direkte Anwendung für Konzept

12.1.2. Modellbasierte Architektur

Versenden der daraus resultierenden Stati -> Indirekte Anwendung für Konzept

12.2. Entscheid

Nachrichtenbasiert -> weniger verbreitet, komplexer in der Implementation -> einfachere Transition.

12.3. Design

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

Die Bausteine werden in den folgenden Kapitel erläutert.

12.3.1. Nachrichten

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

Eine Nachricht wir entsprechend ihrem Ziel und Funktion benannt.

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

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

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

Somit ist implizit eine Mutationsfunktion, und die Referenz auf den Status definiert.

12.3.2. Backend

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

Logik

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

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

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

Persistenz

Das Verhalten des Singlestate Konzepts ist mit einer Datenbank abbildbar.

12.3.3. API

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

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

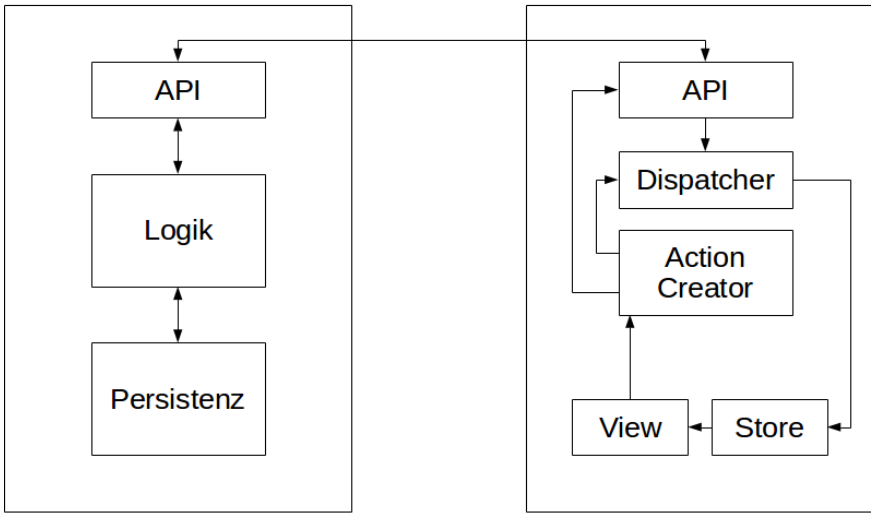
Nachrichtname	Beschreibung
S_API_WEB_get	Die Get-Nachricht gibt eines oder alle Objekte einer Collection zurück.
S_API_WEB_put	Ein neues Objekt wird mit der Put-Nachricht erstellt.
S_API_WEB_update	Mit der Update-Nachricht können bestehende Objekte aktualisiert werden.
S_API_WEB_delete	Bestehende Objekte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

12.3.4. Frontend

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

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

12.3.5. Datenfluss



Frontend <-> API <-> Backend

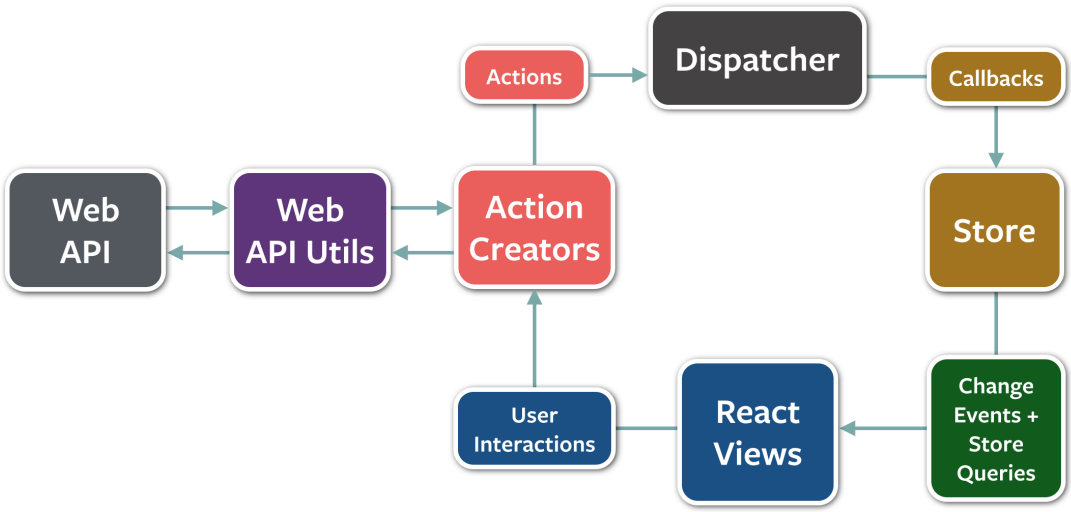
Frontend: ActionCreator -> Dispatcher -> Store -> API -> ActionCreator

API: Queue -> Transporter

Backend: API -> Logiclayer -> API

12.3.6. Flux Architektur

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



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

12.3.7. AMD Pattern

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

12.4. Beispielapplikation

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

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

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

Teil IV.

Implementierung und Testing

13 Prototyp

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

13.1. Umsetzung

13.2. Technologie Stack

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

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

13.3. Entwicklungsumgebung

Grunt + Karma = All you need

13.4. Entwicklung

Tricks mit API & Message Routing, binding to io.on „message“ -> flux.doAction

Express Server:
Statisches Daten -> Frontend /
Socket.io -> /socket.io
Message-Bus -> Fluxify also in the backend

RequireJS Modules testable in the Browser :-D

Stores in the Frontend
Models in the Backend

13.5. Grafische Umsetzung Fallbeispiel

14 Testing

14.1. Unit-Testing

14.2. Integration-Testing

Teil V.

Abschluss und Ausblick

15 Review

15.1. Validation

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

Ziel
R1
R2
R3
R4
R5

15.2. Ausblick

16

Fazit und Schlusswort

A
Appendix

A.1. Glossar

ORM
Node

A.2. Aufgabenstellung

A.2.1. Thema

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

A.2.2. Ausgangslage

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

A.2.3. Ziele der Arbeit

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

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

A.2.4. Aufgabenstellung

A1 Recherche:

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

A2 Analyse:

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

A3 Konzept:

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

A4 Prototyp:

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

A5 Review:

- Test des Prototyps und Protokollierung der Ergebnisse

A.2.5. Erwartete Resultate

R1 Recherche:

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

R2 Analyse:

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

R3 Konzept:

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

R4 Prototyp:

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

R5 Review:

- Protokoll der Tests des Software Prototypen

B

Detailanalyse der Aufgabenstellung

Die Detailanalyse der Aufgabenstellung...

B.1. Aufgabenstellung und erwartete Resultate

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

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

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

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

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

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

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

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

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

Erwartet wird ein Protokoll der durchgeführten Tests.

B.2. Aufwandschätzung

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

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

C

Projektmanagement

C.1. Projektplanung

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

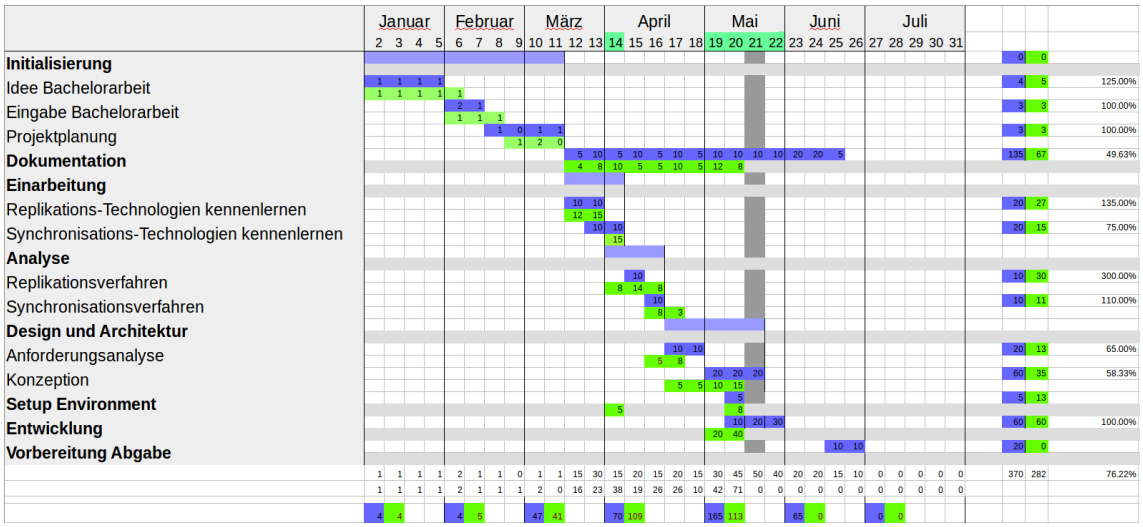


Abbildung C.1.: Projektplan

C.2. Termine

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

Termin	Datum	Bemerkungen
Abschlusspräsentation	??	

C.3. Dokumentation

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

C.4. Versionsverwaltung

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

D

Anforderungsanalyse

D.1. Vorgehensweise

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

D.1.1. Use-Cases

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

UC-01 Lesen eines Elements

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

UC-02 Einfügen eines neuen Elements

UseCase	
Ziel	Ein neues Objekt wird hinzugefügt.

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

UC-03 Ändern eines Elements

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

UC-04 Löschen eines Elements

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

D.1.2. Anforderungen

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

FREQ01.01 Abfragen eines Objektverzeichnis

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

FREQ01.02 Abfragen eines bekannten Objekt vom Server

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

FREQ02.01 Senden eines neuen Objekt

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

FREQ02.02 Abfragen eines neu hinzugefügten Objekt

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

FREQ03.01 Senden einer Objektmutation

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

FREQ03.02 Senden einer Objektmutation

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

FREQ04.01 Löschen eines Objekts

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

Anforderung	

FREQ04.01 Lokale Kopie gelesener Objekte

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

FREQ05.01 Aufzeichnen der Mutationen

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

FREQ05.02 Übermitteln der Mutationen

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

NFREQ01 Übermittlung der Mutationen

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

NFREQ02 Abgelehnte Mutationen

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

D.1.3. Akzeptanzkriterien

D.1.4. Bewertung der Anforderungen

D.2. Risiken

E

Verzeichnisse

E.1. Quellenverzeichnis

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

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

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

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

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

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

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

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

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

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

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

E.2. Tabellenverzeichnis

E.3. Abbildungsverzeichnis