ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Synchronisation und Fehlertoleranz mobiler Clients in einem bestehenden System für kollaboratives Task-Management

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Enrico Weigelt
enrico.weigelt@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Nicolas Mundbrod

2015

Fassung 27. Mai 2015

Satz: PDF-LaTeX 2ε

Kurzfassung

Durch die zunehmende Globalisierung und Digitalisierung wird das Fachwissen der Wissenensarbeit zur Grundlage vieler Prozesse in hoch entwickelten Ländern. Gleichzeitig verlagert sich die Fertigung in Länder mit niedrigerem Lohnniveau. Die Wissensarbeit in der Entwicklung wird in den Prozessen von vielen, auf verschiedene Fachgebiete, spezialisierten Fachkräften kolloborativ verrichtet. Da wissensintensive Prozesse komplex und dynamisch sind, eignen sich bewährte Ansätze und Lösungen für standardisierte Prozesse nicht zur Unterstützung. Eine Möglichkeit der koordinierten Unterstützung von Wissensarbeit ist das Management von Checklisten.

Im Rahmen des Forschungsprojekts proCollab sollen die Möglichkeiten des kolloborativen Checklisten-Managements zur Unterstützung der Wissensarbeit konzipiert und evaluiert werden. Im Konkreten soll Wissensarbeitern, die sich an verschiedenen Orten befinden, ermöglicht werden, sich auch anhand von mobilen Applikationen zu koordinieren. In dieser Arbeit werden der vorhandene proCollab-Prototyp und die mobilen Applikationen deshalb um ein Konzept zum Synchronisationsmechanismus und der Fehlertoleranz erweitert.

Inhaltsverzeichnis

1

Einleitung

In der heutigen vernetzten Welt ist effizienter und schneller Informationsaustausch unerlässlich. Durch die immer weiter fortschreitende Globalisierung und Digitalisierung werden Geschäftsprozesse innerhalb von Unternehmen immer komplexer. Vor allem in hoch entwickelten Industrienationen sind die Beschäftigungszahlen in der Fertigung rückläufig und verlagern sich dagegen in die Bereiche Forschung und Entwicklung [Eic14].

Durch diesen strukturellen Wandel der Industriegesellschaft hin zur Informationsgesellschaft stehen Unternehmen vor ganz neuen Herausforderungen [KS09]. Die Anzahl der Mitbewerber auf dem internationalen Markt wächst, der Wettbewerbsdruck steigt und die Innovationszyklen werden gleichzeitig kürzer. Innovationen müssen in immer kürzeren Abständen erfolgen. Auf der anderen Seite können die Kosten der Produktion durch Verlagerung in Länder mit niedrigerem Lohnniveau gesenkt werden. An den

Entwicklungsstandorten verbleiben komplexe Arbeiten, die ein hohes Maß an Fachwissen erfordern und nicht in einfachen Abläufen planbar sind. Diese wissensintensiven Prozesse werden von vielen verschiedenen Fachkräften mit diversen Spezialisierungen kooperativ angegangen und gelöst. Diese Fachkräfte befinden sich jedoch zwangsläufig nicht am selben Standort.

Die Unternehmen sind stets bestrebt, die Zusammenarbeit der menschlichen Aktivitäten zu optimieren und zu verbessern. Außerdem werden Arbeits- und Entwicklungsmethoden ständig überwacht und bei Bedarf angepasst [Paw98].
Um das kolloborative, verteilte Management von Aufgaben zu unterstützen, wurde das Forschungsprojekt proCollab[1] an der Universität Ulm gestartet [pro]. Das proCollab-Projekt, das aus einer Server Komponente und mobilen Applikationen besteht, ermöglicht das Verwalten von Checklisten, die verschiedene Einträge als Aufgaben für die Wissensarbeit beinhalten. Die Wissensarbeiter können durch mobile Applikationen den Status ihrer Aufgaben abfragen und ändern. Außerdem können sie sich über den Status von Aufgaben anderer Wissensarbeiter informieren.

1.1 Problemstellung

Der bestehende proCollab-Prototyp mit seinen mobilen Applikationen soll erweitert werden, um Wissensarbeiter effektiver zu unterstützen. Das Ziel ist es hierbei, die Verwaltung verteilter Aufgaben zu ermöglichen. Die Synchronisation von Informationen und im Speziellen von Aufgaben über viele mobile Clients hinweg bringt Problemstellen mit sich.
Zum einen müssen die grundlegenden Probleme verteilter Systeme beachtet werden. Da konkurrierende Zugriffe, der Clients auf den Prototyp möglich sein sollen, müssen die Zugriffe auf Konflikte hin überprüft werden. Typischerweise treten diese auf wenn mehrere Clienten die gleichen Daten, z.B. eine Checkliste, bearbeiten, ohne die Änderungen der anderen Clients zuvor synchronisiert zu haben [Boo76]. Da Daten über mehrere Clients verteilt und auf diesen bearbeitet werden sollen, müssen diese kontinu-

[1] Process-aware Support for Collaborative Knowledge Workers

ierlich aktualisiert werden, um Konflikte zu vermeiden. Zusätzlich gibt es Faktoren, die durch die Kommunikation des Clients mit dem Server über ein Netzwerk bedingt sind. Zu nennen sind hier bspw. die Bandbreite der Netzwerkverbindung, sowie Fehler die bei der Übertragung innerhalb des Kommunikationsnetzes auftreten [PL94]. Da diese Probleme unerwartet auftreten können, muss ein schnelles Erkennen der Situation und eine entsprechende Fehlerbehandlung gewährleistet sein, um die Systeme nutzbar zu halten.

1.2 Beitrag

Beitrag dieser Bachelorarbeit ist die Konzeption und Entwicklung eines Verfahrens zur Synchronisation und Fehlerbehandlung für die mobilen Clients des proCollab-Prototypen. Das Verfahren soll implementiert werden und damit die bestehenden mobilen Applikationen für Smartphone [Gei13] und Tablet [Köl13] erweitert werden. Ziel ist eine konsistente und fehlertolerante Synchronisation von Daten im proCollab-Prototyp über Endgeräte hinweg.

Änderungen auf den mobilen Applikationen, die bei nicht vorhandener Internetverbindung durchgeführt wurden, sollen zunächst in der mobilen Applikation bis zu einem gewissen Grad zwischengespeichert werden und im Hintergrund auf den Server übertragen werden, sobald eine Verbindung aufgebaut werden kann. Der Benutzer soll bei Auftreten von oben genannten Konflikten, die sich nicht automatisch lösen lassen, informiert werden und Möglichkeiten zur manuellen Konfliktlösung angeboten bekommen. Die Kommunikation der mobilen Applikationen mit dem proCollab-Prototyp läuft dabei über eine eigene REST-Schnittstelle ab [Zie13], die um ein Algorithmus zur Konfliktlösung erweitert wird.

1.3 Aufbau

Diese Arbeit besteht aus sechs Kapiteln, die jeweils hier kurz zusammengefasst werden. Die einzelnen Grundthemen werden in den Tabellen 1.1 bis 1.5 gezeigt.

In **Kapitel 2** wird das Projekt *proCollab* vorgestellt. Es werden die Architektur und möglichen Anwendungsgebiete erläutert. Außerdem wird das Datenmodell beschrieben.

Wissensintesive Prozesse	Überblick	Anwendungsgebiete	Architektur	Datenmodell

Tabelle 1.1: Übersicht: Kapitel 2

Kapitel 3 dient der Grundlagenbildung, dabei werden grundlegende Konzepte und Probleme, die in verteilten Systemen auftreten können, erläutert. Außerdem werden Mechanismen, die in verteilen Systemen verwendet werden, vorgestellt.

CAP-Theorem	Nebenläufigkeit in verteilen Systemem	Mechanismen verteilter Systeme

Tabelle 1.2: Übersicht: Kapitel 3

Anschließend werden in **Kapitel 4** die Qualitätskriterien und Anforderungen dargestellt. Dazu werden Anwendungsfälle erstellt und diese anschließend analysiert.

Allgemein	Anwendungsfälle	Analyse der Anwendungsfälle

Tabelle 1.3: Übersicht: Kapitel 4

In **Kapitel 5** wird auf Basis der vorangegangenen Kapitel ein Synchronisation-Konzept erarbeitet. Dieses muss die Anforderungen, die bei den Anwendungsfällen erstellt wurden, erfüllen.

CAP	Verwendete Mechanismen	Konfliktlösung	Durchführung

Tabelle 1.4: Übersicht: Kapitel 5

Im **Kapitel 6** wird anschließend die eigentliche Implementierung besprochen. Hierbei werden zunächst die genutzten Technologien eingeführt und anschließend Auszüge der Implementierung auf Client- und Serverseite angegeben.

Datenmodell	Verwendete Technologien	Implementierung Server	Implementierung Client

Tabelle 1.5: Übersicht: Kapitel 6

Im letzten Kapitel werden die Ergebnisse zusammengefasst und ein kleiner Ausblick gegeben.

Zusammenfassung	Ausblick

Tabelle 1.6: Übersicht: Kapitel 7

2

proCollab

2.1 Wissensintensive Prozesse

Durch die voranschreitende Globalisierung verlagern sich immer mehr automatisierbare Prozesse ins Ausland, um zum Beispiel von günstigen Lohnkosten zu profitieren. Im Gegensatz zu automatisierbaren Prozessen, wie die Fertigung eines Bauteils, sind wissensintensive Prozesse nicht standardisierbar. Sie benötigen ein hohes Fachwissen oder sind zu komplex, um automatisiert zu werden. Diese Prozesse verbleiben deshalb in Ländern mit höherem Bildungsniveau [PS12].

Typisch für solche wissensintensiven Prozesse sind viele Einflussfaktoren, wie zum Beispiel eine stetig wachsende Wissensbasis während des Verlaufs oder die schrittweise Entstehung des Arbeitsablaufs [MKR13]. Um diese Prozesse möglichst effizient bearbeiten zu können, arbeiten meist mehrere Wissensarbeiter zusammen. Diese

Zusammenarbeit wird auch als kollaborative Wissensarbeit bezeichnet, die sich aus verschiedenen wissensintensiven Prozessen zusammensetzt. Da diese Prozesse eine hohe Dynamik und Schnelllebigkeit entwickeln, müssen sie ständig auf Basis ihres aktuellen Zustands überprüft, sowie auf Veränderungen schnell reagiert werden. Da die Prozesse durch Wissensarbeit getrieben werden, stellen diese gleichzeitig auch die größte Fehlerquelle dar. Allerdings können Fehler durch geeignete Unterstützungen, wie zum Beispiel Checklisten, verhindert werden (siehe Kapitel 2.3).

2.2 Überblick über proCollab

proCollab ist ein Forschungsprojekt an der Universität Ulm [MR14], das sich damit beschäftigt Lösungen zu entwickeln, die Wissensarbeiter effizient in wissensintensiven Prozessen unterstützen. Der Kommunikations- und Kooperationsaufwand soll dabei optimiert werden. Eine mögliche Lösung, die in vergangenen Arbeiten evaluiert wurde, ist das Checklisten-Management. Für diesen Zweck wurden einzelne Prototypen für Smartphones und Tablets erstellt, welche im Folgenden proCollab-Clients (pCC) genannt werden. Die nötigen Schnittstellen und die Anwendungslogik auf Serverseite, im Folgenden proCollab-Prototyp (pCP) genannt, wurden in separaten Arbeiten realisiert [Zie13] [Rei13] [Thi13]. Der Beitrag dieser Arbeit wird auf Basis der vorhandenen Arbeiten erstellt. In Abbildung 2.1 werden schematisch die verschiedenen Komponenten von proCollab dargestellt.

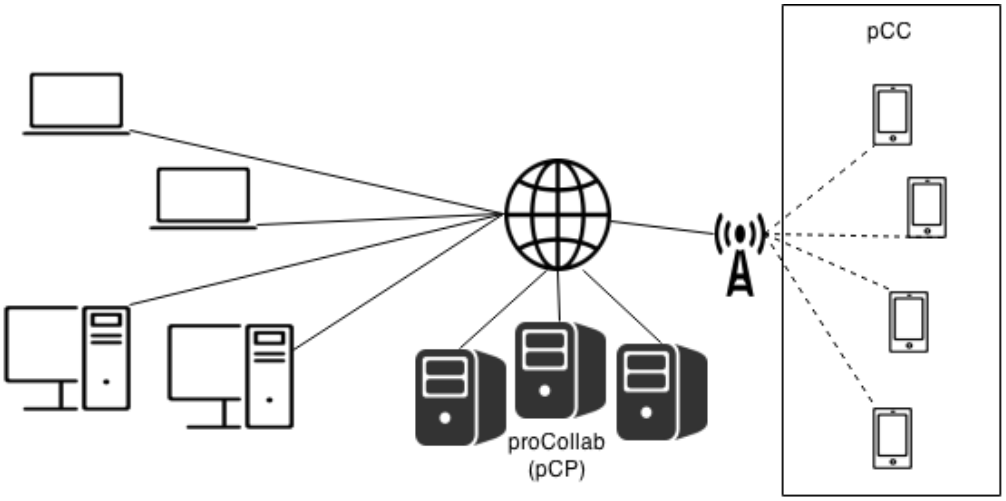


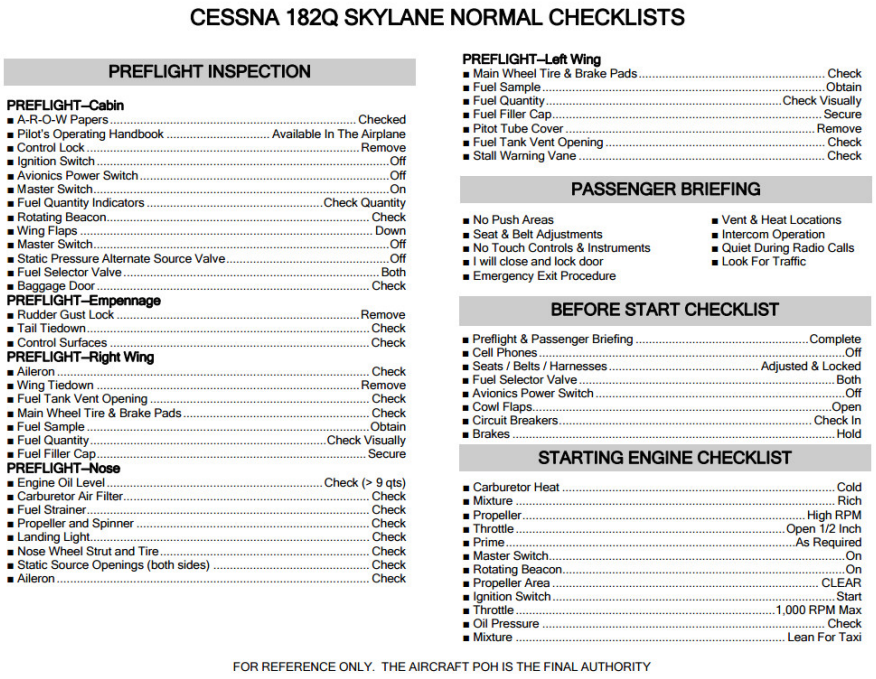
Abbildung 2.1: Schematische Darstellung der proCollab Komponenten

2.3 Anwendungsgebiete

Aufgabenverwaltung in Form von Checklisten wird heute schon in vielen Bereichen eingesetzt, um Arbeitsabläufe zu planen oder Fehler zu vermeiden. Zum besseren Verständnis des Checklisten-Managements werden nun vier Beispiele aus der Berufswelt gezeigt.

2.3.1 Luftfahrt

In der **Luftfahrt** können schon kleine technische Probleme oder menschliche Fehler große Auswirkung haben. So gab es in den Anfängen der Luftfahrt eine Vielzahl von Abstürzen, die auf menschliches Versagen oder fehlerhafte Wartung zurückzuführen waren [Joh]. Um dem entgegen zu wirken, existieren für nahezu alle Arbeiten, die in und um ein Flugzeug erledigt werden müssen, Checklisten, nach denen vorgegangen werden muss. So darf zum Beispiel erst gestartet werden, wenn alle Punkte des Preflight-Checks erfolgreich abgearbeitet wurden. In Abbildung 2.2 sind vier der 18 Checklisten für eine Cessna 182 angegeben. Der Nutzen von Checklisten wird auch an den seit Jahren sinkenden Unfallraten in der kommerziellen Luftfahrt deutlich [ICA14].



CESSNA 182Q SKYLANE NORMAL CHECKLISTS

PREFLIGHT INSPECTION

PREFLIGHT–Cabin
- A-R-O-W Papers ..... Checked
- Pilot's Operating Handbook ..... Available In The Airplane
- Control Lock ..... Remove
- Ignition Switch ..... Off
- Avionics Power Switch ..... Off
- Master Switch ..... On
- Fuel Quantity Indicators ..... Check Quantity
- Rotating Beacon ..... Check
- Wing Flaps ..... Down
- Master Switch ..... Off
- Static Pressure Alternate Source Valve ..... Off
- Fuel Selector Valve ..... Both
- Baggage Door ..... Check

PREFLIGHT–Empennage
- Rudder Gust Lock ..... Remove
- Tail Tiedown ..... Check
- Control Surfaces ..... Check

PREFLIGHT–Right Wing
- Aileron ..... Check
- Wing Tiedown ..... Remove
- Fuel Tank Vent Opening ..... Check
- Main Wheel Tire & Brake Pads ..... Check
- Fuel Sample ..... Obtain
- Fuel Quantity ..... Check Visually
- Fuel Filler Cap ..... Secure

PREFLIGHT–Nose
- Engine Oil Level ..... Check (> 9 qts)
- Carburetor Air Filter ..... Check
- Fuel Strainer ..... Check
- Propeller and Spinner ..... Check
- Landing Light ..... Check
- Nose Wheel Strut and Tire ..... Check
- Static Source Openings (both sides) ..... Check
- Aileron ..... Check

PREFLIGHT–Left Wing
- Main Wheel Tire & Brake Pads ..... Check
- Fuel Sample ..... Obtain
- Fuel Quantity ..... Check Visually
- Fuel Filler Cap ..... Secure
- Pitot Tube Cover ..... Remove
- Fuel Tank Vent Opening ..... Check
- Stall Warning Vane ..... Check

PASSENGER BRIEFING
- No Push Areas
- Seat & Belt Adjustments
- No Touch Controls & Instruments
- I will close and lock door
- Emergency Exit Procedure
- Vent & Heat Locations
- Intercom Operation
- Quiet During Radio Calls
- Look For Traffic

BEFORE START CHECKLIST
- Preflight & Passenger Briefing ..... Complete
- Cell Phones ..... Off
- Seats / Belts / Harnesses ..... Adjusted & Locked
- Fuel Selector Valve ..... Both
- Avionics Power Switch ..... Off
- Cowl Flaps ..... Open
- Circuit Breakers ..... Check In
- Brakes ..... Hold

STARTING ENGINE CHECKLIST
- Carburetor Heat ..... Cold
- Mixture ..... Rich
- Propeller ..... High RPM
- Throttle ..... Open 1/2 Inch
- Prime ..... As Required
- Master Switch ..... On
- Rotating Beacon ..... On
- Propeller Area ..... CLEAR
- Ignition Switch ..... Start
- Throttle ..... 1,000 RPM Max
- Oil Pressure ..... Check
- Mixture ..... Lean For Taxi

FOR REFERENCE ONLY. THE AIRCRAFT POH IS THE FINAL AUTHORITY

Abbildung 2.2: Checklisten für Cessna 182, [ces]

2.3.2 Medizin

Ein anderes Anwendungsgebiet für Checklisten-Management ist die **Medizin**. Hier ist ein guter Informationsaustausch zwischen den verschiedenen Berufsgruppen und Zuständigkeiten unerlässlich. Es können dadurch Fehler vermieden werden. Ein konkretes Beispiel sind medizinische Visiten, bei denen viele Parameter erfasst und aktualisiert werden müssen. Dabei braucht der Arzt einen Überblick über schon verabreichte Medikamente und eventuelle Laborergebnisse. Außerdem müssen Anweisungen zur weiteren Behandlung für andere Mitarbeiter bekannt gemacht werden. Dies wird heute noch oft in Papierform erledigt. Hier besteht eine Einsatzmöglichkeit für mobile Applikationen, die mit dem MEDo- Ansatz evaluiert wurde [PMLR13]. Die Aufgabenverwaltung von MEDo wird in Abbildung 2.3 illustriert. Auch die **Alten-** und **Krankenpflege** sind mögliche Einsatzgebiete.

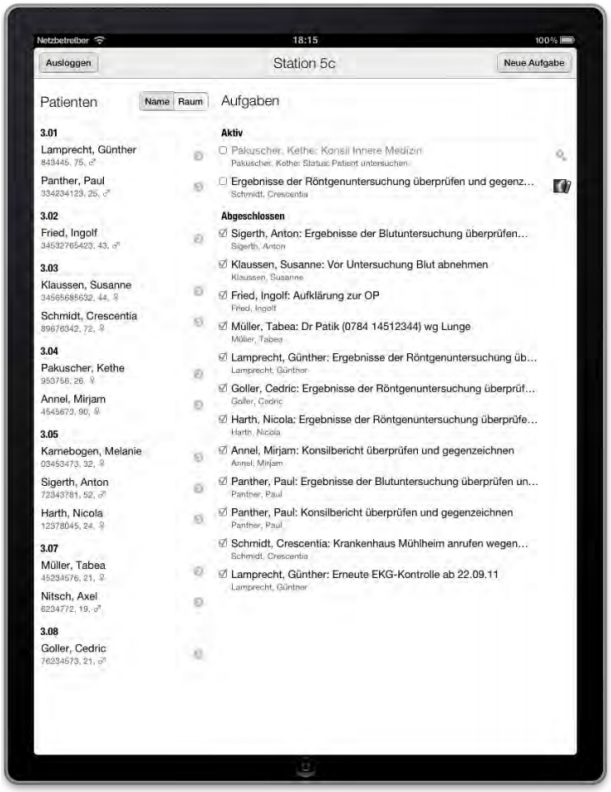


Abbildung 2.3: Aufgabenverwaltung in MEDo, [Lan12]

2.3.3 Lebensmittelkontrolle

Um bei Lebensmitteln eine gleichbleibende Qualität zu gewährleisten und Gesundheitsgefährdungen zu vermeiden, gelten hier hohe Qualitätsansprüche. So müssen zum einen die Produktions- und Lagerstätten bestimmte hygienische Anforderungen erfüllen (Abbildung 2.4). Zum anderen müssen für die verschiedenen Lebensmittel optimale Lagerbedingungen erfüllt sein. Um die nötigen Grenzwerte zu überprüfen und zu dokumentieren, werden auch hier Checklisten verwendet [Bun03]. Durch diese Dokumentation können bei Schwankungen in der Qualität schnell mögliche Fehlerquellen lokalisiert werden.

MODUL 3 - UMSETZUNG HYGIENE UND SCHULUNG **FRIST**

M 3/1	Optische Beurteilung der Qualität der Reinigung in Räumen und Bereichen sowie von Oberflächen.	
M 3/2	Optische Beurteilung der Qualität der Reinigung von Gebrauchsgegenständen, Anlagen und Maschinen.	
M 3/3	Optische Beurteilung der Qualität der Reinigung von Transportbehältern und -fahrzeugen.	
M 3/4	Optischer Zustand und Lagerung von Reinigungsutensilien, Putz- und Desinfektionsmitteln.	
M 3/5	Optische Beurteilung der Qualität der Schädlingsbekämpfung in Räumen, in welchen Lebensmittel be- und/oder verarbeitet oder gelagert werden.	
M 3/6	Personalhygiene ist entsprechend (Arbeitskleidung, Kopfbedeckung, kein Tragen von Schmuck, keine Hautausschläge und offenen Wunden ...).	
M 3/7	Optische Beurteilung der Qualität der Reinigung von Personalräumen (Personaltoiletten, Umkleide- und Waschräume) und Bereitstellung sauberer Arbeitskleidung.	
M 3/8	Überprüfung der Wirksamkeit der Schulung.	
In ALIAS einzugebende Bewertung für das Modul 3 ***(Umsetzung Hygiene und Schulung)***		

Referenzdokument: VA-K – 0232 (Verfahrensanweisung) Version 3.0
Qualitätsmanagementhandbuch der Lebensmittelaufsicht Österreichs

Abbildung 2.4: Hygienecheckliste der Lebensmittelkontrolle, [leh11]

2.3.4 Softwareentwicklung

Auch in der Softwareentwicklung werden Checklisten und Abwandlungen dessen zur Aufgabenverwaltung verwendet. Die einzelnen Aufgaben sind dabei meist, wie in Abbildung 2.5 gezeigt, mit einem Status versehen. So ist für alle in das Projekt involvierten Wissensarbeiter zu jeder Zeit sichtbar, welche Aufgaben noch zu erledigen sind, sich in Bearbeitung befinden oder bereits fertiggestellt sind.
Wird durch den Verlauf der Entwicklung eine neue Aufgabe entdeckt, kann diese direkt dem spezialisierten Wissensarbeiter zugewiesen werden. Über die Priorität kann außerdem die Dringlichkeit der Aufgabe angegeben werden. Dies ist vor allem dann hilfreich, wenn andere Aufgaben durch diese blockiert werden können.
Mehrere Teilaufgaben werden dabei oft zu sogenannten Milestones zusammengefasst.

Dadurch ist eine schrittweise Entwicklung möglich, wobei einzelne Teillösungen regelmäßig überprüft und mit den Anforderungen verglichen werden können [RFG13].

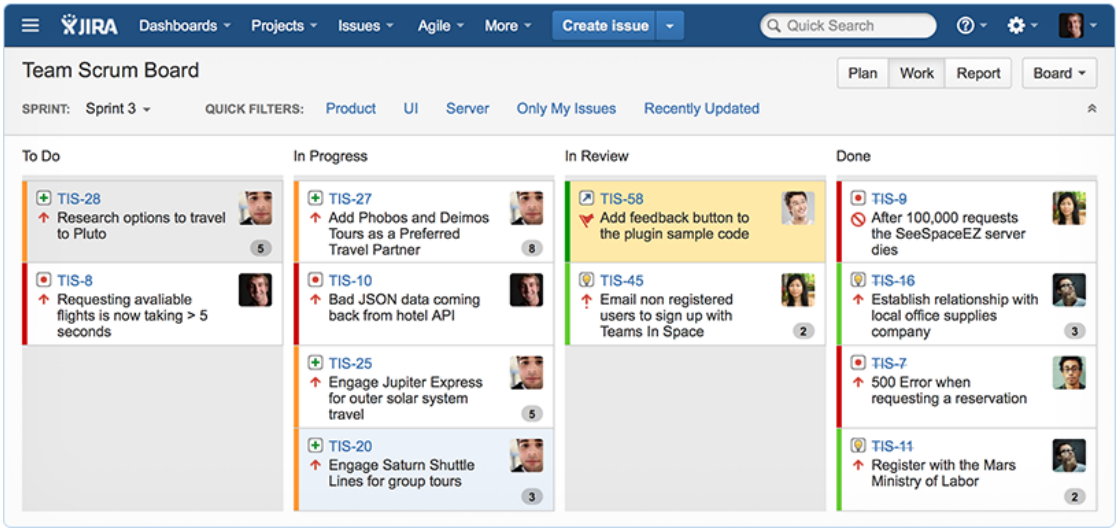


Abbildung 2.5: Aufgabenverwaltung in JIRA, [jir]

Dies sind natürlich nur einige Beispiele für Anwendungen von Checklisten im alltäglichen Berufsleben. Das Checklisten-Management ist prinzipiell für alle Bereiche anwendbar, in denen Informationen und deren Status mit mehreren Teilnehmern geteilt und bearbeitet werden soll.

2.4 Architektur des proCollab-Prototypen

Der pCP wurde als Mehrschichtenanwendung konzipiert [Kay08]. Die einzelnen Schichten, die in Abbildung 2.6 angegeben werden, beschreiben dabei verschiedene Teilaspekte der Anwendung. Die Präsentationsschicht wird in der Grafik zum besseren Verständnis mit angegeben, befindet sich aber auf der Clientseite. Die grau markierten Komponenten werden der Vollständigkeit wegen angegeben, in dieser Arbeit aber nicht weiter besprochen. Dies ist ein gängiges Modell zur Architektur von Softwaresystemen, da dadurch die Komplexität der einzelnen Komponenten reduziert wird, sowie die Ent-

wicklung und Wartbarkeit vereinfacht wird. Innerhalb der Schichten ist der pCP in weitere kleine Teilpakete unterteilt.

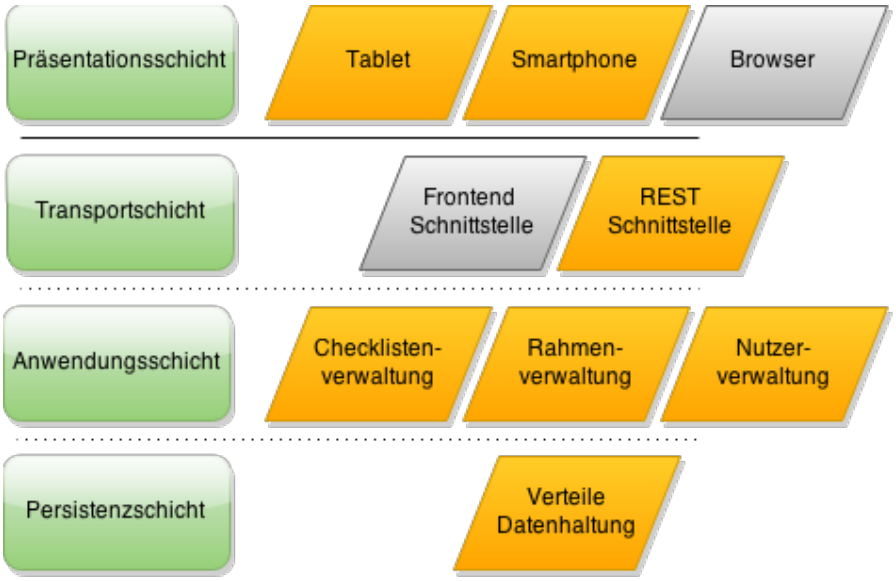


Abbildung 2.6: Applikationsschichten des pCP

Präsentationsschicht Die Präsentationsschicht, repräsentiert die verschiedenen Schnittstellen für Endgeräte, die mit dem pCP verwendet werden können. Die Komponenten Tablet, Smartphone und Webbrowser stellen dabei verschiedene, für das Endgerät optimierte grafische Oberflächen zu Verfügung. Die Präsentationsschicht ist auf dem Client vorhanden. Der pCP liefert lediglich Schnittstellen zur Kommunikation (siehe Transportschicht).

Transportschicht Innerhalb der Transportschicht befinden sich mehrere Komponenten, die die Kommunikation mit der eigentlichen Anwendungslogik über verschiedene Technologien ermöglichen. Die REST-Schnittstelle wird dabei für die Kommunikation mit dem pCC verwendet.

Anwendungsschicht In der Anwendungsschicht befindet sich die Anwendungslogik des pCP. Diese Schicht kann über mehrere Server verteilt werden.

Persistenzschicht Innerhalb der Persistenzschicht werden die Daten gespeichert. Diese kann dabei, wie die Anwendungsschicht, über mehrere Server verteilt werden.

2.5 Architektur des proCollab-Clients

Auch die mobile Applikation, der pCC wurde in mehreren Schichten, angelehnt an das Model-View-Controler-Model (MVC), implementiert [KP88]. Die Schichten des pCC unterteilen sich in View, Controller und Core. Die Hierarchie der Schichten und der schematische Kommunikationsverlauf mit dem pCP werden in Abbildung 2.7 gezeigt.

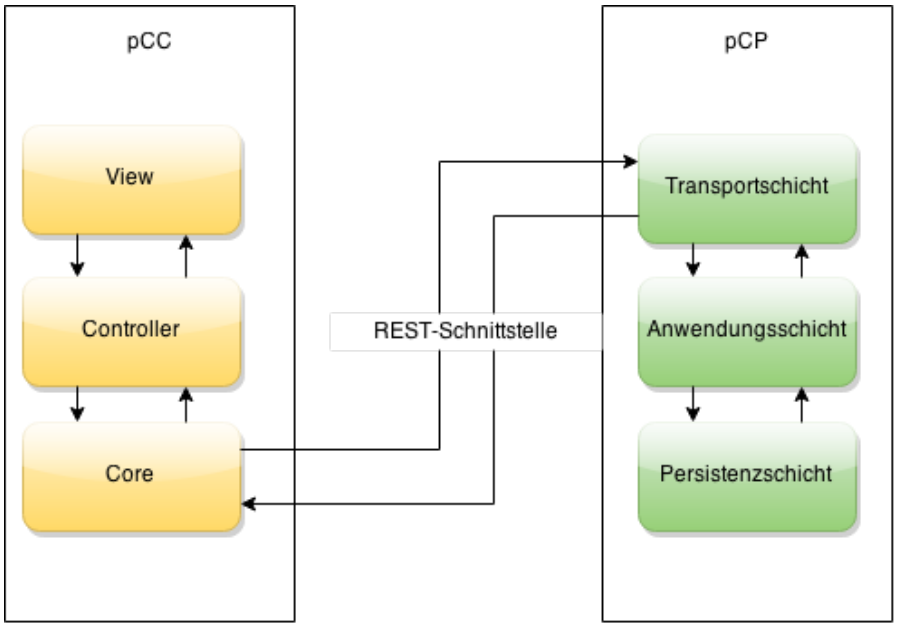


Abbildung 2.7: Applikationsschichten des pCC

View In der Viewschicht wird, wie bei klassischen MVC, das Aussehen der Benutzeroberfläche definiert.

Controller Der Controller dient als Schnittstelle zwischen der View und dem Core. Wird ein Ereignis auf der Benutzeroberfläche ausgelöst, wird dieses vom Controller behandelt.

Core Im Core befinden sich verschiedene Komponenten, die vom Controller zum Verwalten der Daten genutzt werden. Außerdem wird hier die Kommunikation mit der REST-Schnittstelle auf Serverseite realisiert.

2.6 Datenmodell

Innerhalb von *proCollab* gibt es drei grundlegende Datenobjekte. Das Datenmodell ist dabei, wie in Abbildung 2.8 angegeben, hierarchisch aufgebaut.

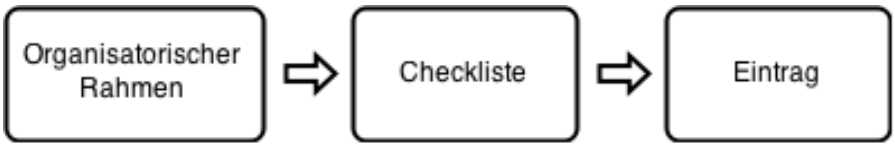


Abbildung 2.8: Hierarchisches Datenbankmodell

2.6.1 Organisatorischer Rahmen

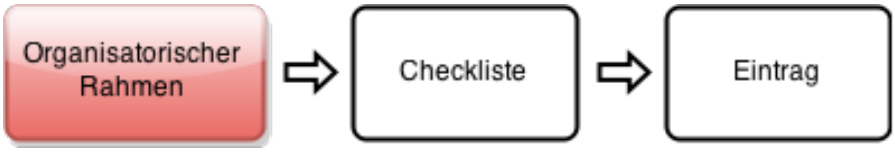


Abbildung 2.9: Hierarchisches Datenbankmodell: Organisatorischer Rahmen

Der *Organisatorische Rahmen* definiert einige grundlegende Eigenschaften eines Projektes innerhalb von *proCollab* . Die relevanten Felder des Rahmens sind in Abbildung 2.10 angegeben. Er gilt somit als Kontext für alle untergeordneten Listen und Einträge. Als Beispiel dient hier der verantwortliche Verwalter , der für alle Elemente unterhalb des

organisatorischen Rahmens gilt. Der organisatorische Rahmen ist dabei selbst in zwei verschiedene Datenmodelle aufgeteilt. So gibt es zum einen den Rahmen*typ* und zum anderen die Rahmen*instanz*.

Der *Rahmentyp* ist dabei eine Vorlage mit allgemeingültigen Eigenschaften, die beliebig angepasst und geändert werden können. Durch diese mögliche Wiederverwendung wird die Zeit zum Erstellen von *proCollab* -Projekten minimiert. Ein Rahmentyp kann Checklistentypen als Kinder enthalten.

Die *Rahmeninstanz* ist das eigentliche Projekt, das unter Verwendung eines Rahmentyps oder von individuellen Parametern erstellt wurde. Innerhalb eines Rahmen sind ein oder mehrere Checklisteninstanzen enthalten.

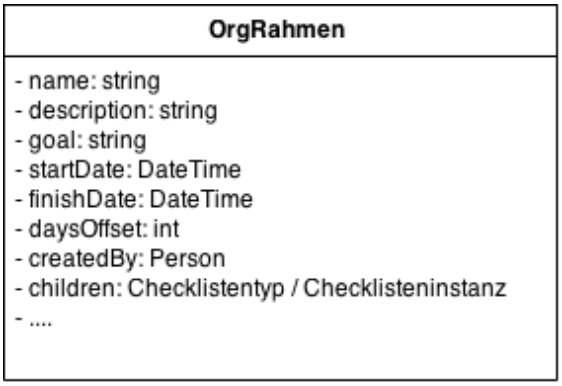


Abbildung 2.10: Datenmodell: Organisatorischer Rahmen

2.6.2 Checkliste

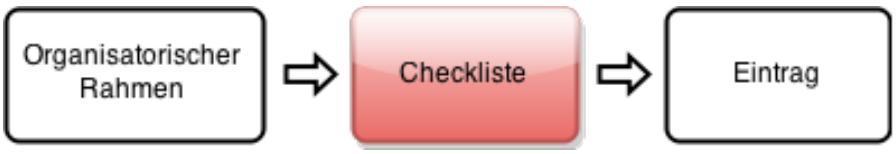


Abbildung 2.11: Hierarchisches Datenbankmodell: Checkliste

Die *Checkliste* ist immer einem organisatorischen Rahmen untergeordnet und enthält mindestens einen Eintrag. Außerdem werden über Checklisten die Reihenfolgen von

Einträgen definiert. Auch bei der Checkliste gibt es wieder zwei Typen. Man spricht von *Checklistentyp* und von *Checklisteninstanz*. Die Attribute der Checkliste sind in Abbildung 2.12 gezeigt.

Der *Checklistentyp* ist Teil von einem Rahmentyp und stellt eine Vorlage für eine Checklisteninstanz dar, die möglicherweise schon einige Eintragstypen enthält oder andere Eigenschaften vordefiniert hat.

Die *Checklisteninstanz* dagegen ist wieder, wie schon bei der Rahmeninstanz, eine Checkliste in Nutzung. Diese ist Teil einer Rahmeninstanz und wurde entweder auf Basis eines Checklistentyps oder von individuellen Parametern erstellt. Sie enthält wiederum Eintragsinstanzen.

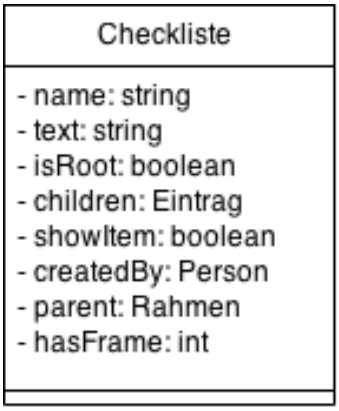


Abbildung 2.12: Datenmodell:Checkliste

2.6.3 Eintrag

Abbildung 2.13: Hierarchisches Datenbankmodell: Eintrag

Ein *Eintrag* enthält in *proCollab* die für die Aufgabe relevanten Informationen . Diese befindet sich immer innerhalb von Checklisten und haben einen Status. Darüber lässt sich prüfen, ob ein Eintrag erledigt ist oder noch Arbeiten daran nötig sind. Auch der Eintrag verfügt über die beiden bekannten Typen *Eintragstyp* und *Eintragsinstanz*.

Ein *Eintragstyp* beschreibt eine Vorlage für einen Eintrag und ist einem Checklistentyp untergeordnet.

Die *Eintragsinstanz* ist ein aus Parametern oder über einen Eintragstyp erstellter Eintrag, der sich innerhalb einer Checklisteninstanz befindet und einen Status aufweist. Ist der Staus auf *FINISHED* gesetzt, kann der Eintrag nicht mehr bearbeitet werden. Die relevanten Attribute eines Eintrags sind in Abbildung 2.14 illustriert.

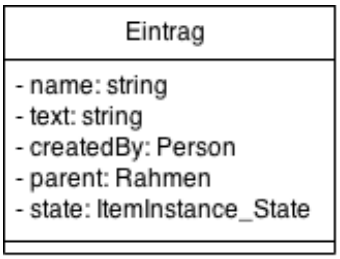


Abbildung 2.14: Datenmodell: Eintrag

2.6.4 Benutzerrollen

Die Benutzerverwaltung des pCP ist in mehrere Rollen mit verschiedenen Berechtigungen unterteilt. Die Benutzerrollen sind hierarchisch aufgebaut (Abbildung 2.15). Dies bedeutet im Speziellen, dass Berechtigungen vererbt werden. Dadurch verfügt beispielsweise der Verwalter über alle Rechte eines normalen Benutzers.

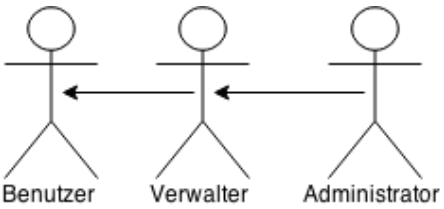


Abbildung 2.15: Nutzerrollen

Benutzer

Der Anwenderrolle *Benutzer* ist die Standardrolle für angemeldete Benutzer. Ein Benutzer ist dabei an einer oder mehreren organisatorischen Rahmeninstanzen beteiligt und bearbeitet ihm zugeteilte Einträge innerhalb einer Checklisteninstanz.

Verwalter

Die Anwenderrolle *Verwalter* verfügt über dieselben Rechte wie der Benutzer. Darüber hinaus kann der Verwalter neue Typen und Instanzen von Checklisten und Einträgen anlegen.

Administrator

Der *Administrator* verfügt über alle Rechte des Verwalters und des Benutzers. Außerdem ist er für die Verwaltung der Benutzer und die organisatorischen Rahmentypen zuständig. Diese Funktionen können allerdings nur über das Webinterface genutzt werden und sind im pCC nicht vorhanden [Thi13].

2.7 IST-Zustand

Im aktuellen Zustand können die pCCs bereits mit dem pCP über die vorhandene REST-Schnittstelle Daten austauschen. Dabei sind jedoch noch keine bekannten Mechanismen vorhanden, die Konsistenzprüfungen oder Ähnliches durchführen. Änderungen werden direkt in die Datenbank eingetragen. Dadurch können ältere Änderungen verloren gehen oder inkonsistente Zustandsübergänge ausgelöst werden. Außerdem weist der pCC aktuell keine Möglichkeit auf zu überprüfen, ob seine lokal vorhandenen Daten noch aktuell sind. Über vollzogene Änderungen wird der Benutzer ebenfalls noch nicht informiert.

Auf dieser Basis ist es notwendig, dass der pCP und pCC um ein Konzept zur Synchronisation und Fehlertoleranz erweitert wird. Dazu werden im Folgenden zunächst Grundlagen verteilter Systeme besprochen.

3

Grundlagen verteilter Systeme

Um ein Konzept zur Synchronisation und Fehlertoleranz des pCP mit den verteilten pCCs zu entwickeln, bedarf es der Betrachtung von Grundlagen verteilter Systeme. Der parallele und verteilte Zugriff von Clients auf eine Komponente in einem verteilen System im Allgemeinen bringt einige Probleme mit sich: Auf der einen Seite dürfen durch parallele Zugriffe konkurrierende Änderungen nicht verloren gehen oder überschrieben werden. Auf der anderen Seite soll das System trotz möglicher Prüfungen verfügbar und benutzbar bleiben.
Auf den folgenden Seiten werden deshalb einige klassische Konzepte und Mechanismen zur Verwaltung von parallelen und verteilten Zugriffen vorgestellt und diskutiert.

3.1 CAP-Theorem

Das *CAP-Theorem* beschreibt die Wechselwirkung zwischen den Eigenschaften *Konsistenz*, *Verfügbarkeit* und *Partitionstoleranz* innerhalb eines verteilten Systems mit n-Knoten. Als Knoten werden dabei im System vorhandene Komponenten wie Server oder Clients bezeichnet. Das CAP-Theorem besagt im Speziellen, dass von den genannten Eigenschaften nie alle drei gleichzeitig erfüllt werden können [Bre12]. Im Jahr 2002 konnte dies von Gilbert und Nancy Lynch bewiesen werden [GL02]. Das CAP-Theorem wird in Abbildung 3.1 veranschaulicht. Der Schnitt aller drei Eigenschaften ist als nicht erreichbar markiert. Die Eigenschaften sind folgendermaßen definiert:

- **Konsistenz**
  Als Konsistenz wird die Korrektheit der Daten verstanden. Diese Korrektheit kann durch Integritätsbedingungen definiert werden. Im Falle des CAP-Theorems bedeutet Konsistenz, dass alle Knoten dieselben Daten zu einen bestimmten Zeitpunkt zur Verfügung haben. Die Konsistenz ist strikt, wenn Änderungen sofort auf allen Knoten verfügbar sind. Sie ist niedrig, wenn ein gewisses Zeitfenster benötigt wird um die Daten zu verteilen. Man spricht in diesem Zusammenhang von *eventual consistency*.

- **Verfügbarkeit**
  Verfügbarkeit bedeutet, dass Anfragen an das System zu jeder Zeit beantwortet werden. Die Verfügbarkeit ist hoch, wenn die Anfragen schnell beantwortet werden und niedrig, wenn es zu Wartezeiten kommt.

- **Partitionstoleranz**
  Partitionstoleranz bedeutet, dass einzelne Knoten des Systems ausfallen können ohne, dass der Betrieb der restlichen Knoten beeinträchtigt wird oder Daten verloren gehen. Innerhalb eines verteilten System sollte diese letzte Eigenschaft immer erfüllt sein.

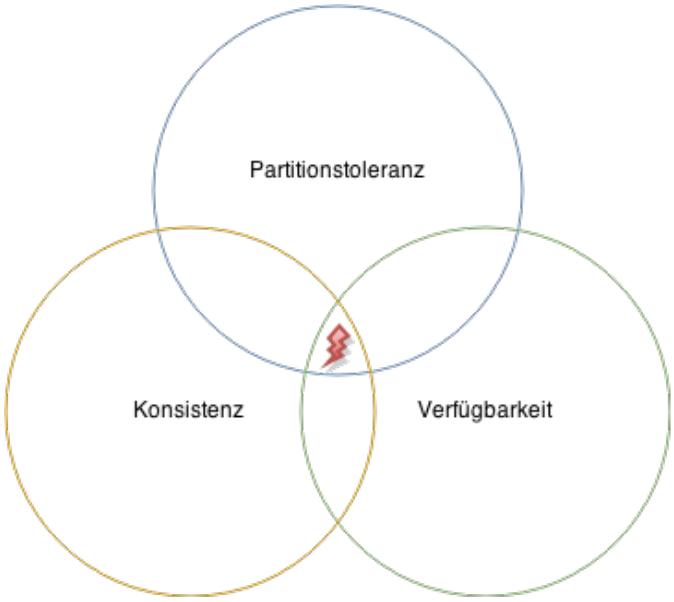


Abbildung 3.1: CAP-Theorem

Aus dem CAP-Theorem geht hervor, dass immer höchstens zwei Eigenschaften erfüllt werden können. Um dies zu verdeutlichen, wird im Folgenden für jeden möglichen Schnitt der Eigenschaften ein Beispiel aus der Realität angegeben. Nicht alle sind für verteilte Systeme sinnvoll. In Abbildung 3.2 sind die relevanten Anwendungsfälle blau markiert.

Konsistenz und Verfügbarkeit: Ein System aus verteilten relationalen Datenbank-Management-Systemem (RDBMS) wie MySQL [MyS], dienen als Beispiel für ein System, das konsistent und verfügbar ist. Im Vordergrund steht, dass die Daten schnell verfügbar und konsistent sind. Bei einem Ausfall einzelner Knoten in einem verteilen RDBMS, ist der fehlerfreie Betrieb nicht gegeben.

Konsistenz und Partitionstoleranz: Ein verteiltes System aus Bankautomaten und zentralem Buchungssystem kann hier als Beispiel genannt werden. Es kommt vor allem darauf an, dass Buchungen verlässlich und korrekt durchgeführt werden können. Fällt ein Bankautomat aus, sind an diesem keine Abbuchungen oder Einzahlungen mehr möglich.

Das restliche System ist aber weiterhin verfügbar und die Daten bleiben konsistent. Ein solches System kann durch ein speziell konfiguriertes verteiltes RDBMS realisiert werden.

Verfügbarkeit und Partitionstoleranz: Als klassisches Beispiel für ein System, das verfügbar und partitionstolerant ist, gilt das *Domain Name System* (DNS) [GL12]. Die Verfügbarkeit des Dienstes zur Auflösung von URLs auf IP-Adressen ist enorm wichtig, da die Adressierung von Webseiten und Services im Internet sich auf diesen Dienst verlassen. Sind einzelne Knoten des DNS nicht erreichbar, übernimmt ein anderer Knoten. Aufgrund der hohen Verfügbarkeit und Partitionstoleranz, sind die Daten jedoch unter Umständen nicht aktuell. Es kann bis zu 24 Stunden dauern, bis sich Änderungen im gesamten DNS-System verteilt haben.

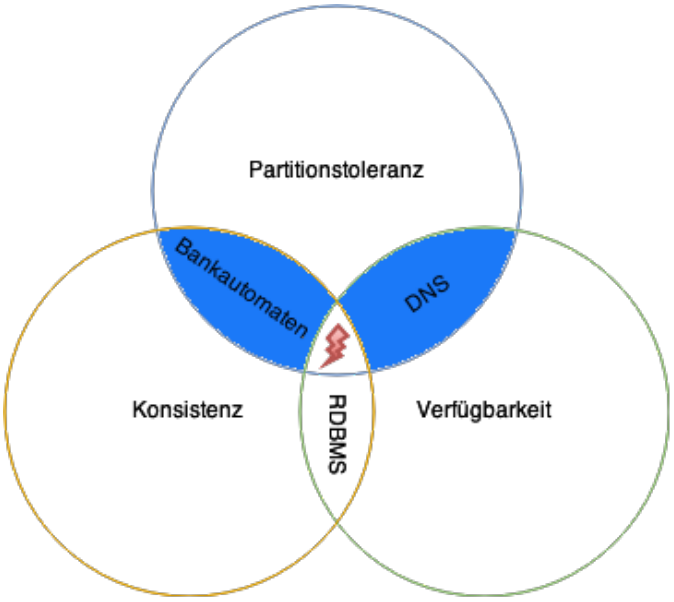


Abbildung 3.2: CAP-Theorem: Anwendungen

Zusammengefasst bedeutet dies, dass man bei verteilten Systemen zwischen Konsistenz und Verfügbarkeit entscheiden muss, da die Partitionstoleranz als Definition für ein verteiltes System gesehen werden kann [Vog07].

3.2 Nebenläufigkeit in verteilten Systemen

Die Nebenläufigkeit in verteilten Systemen bringt, wie eingangs schon erwähnt, einige Probleme mit sich. Diese können durch Konzepte wie ACID und BASE gelöst und minimiert werden.

3.2.1 ACID

ACID ist eine Abkürzung für die gewünschten Eigenschaften Atomarität, Konsistenzerhaltung, Isolation und Dauerhaftigkeit in Datenbanksystemen und verteilten Systemen[1]. Werden diese Eigenschaften erfüllt, kann von einem verlässlichen System gesprochen werden, das sich zu jedem Zeitpunkt in einem konsistenten Zustand befindet und nach Änderungen durch den Benutzer in einen solchen übergeht [HR99]. Die Eigenschaften sind folgendermaßen definiert.

Atomarität

Alle Änderungen am System, auch wenn diese mehrere Schritte umfassen, müssen für den Benutzer atomar sein. Wenn es innerhalb eines Schrittes zu einem Fehler kommt, der die Ausführung abbricht, müssen alle schon getätigten Änderungen zurückgenommen werden. Um dies zu ermöglichen, wird in den meisten Fällen auf *Logging* [HR99] zurückgriffen, mit dessen Hilfe die getätigten Änderungen nachvollzogen und zurückgesetzt werden können. Dieses Verhalten wird auch *Rollback* genannt.

Konsistenzerhaltung

Das System soll sich zu jedem Zeitpunk in einem konsistenten Zustand befinden. Das System muss vor und nach Transaktionen jeweils prüfen, ob die Konsistenz innerhalb der Daten noch erhalten ist. Als Transaktion betrachtet man eine Folge von Abfragen die als logische Einheit betrachtet werden. Dafür könnten zum Beispiel bestimmte Felder

[1] **A**tomicity **C**onsistency **I**solation **D**urability

auf ihren Wertebereich hin überprüft werden oder Schlüsselwerteigenschaften müssen erfüllt sein [HR99]. Wenn diese Eigenschaften nicht zutreffen, wird die Transaktion abgelehnt und wie bei der Eigenschaft *Atomarität* werden schon getätigte Änderungen zurückgenommen. Zwischen den einzelnen Schritten einer Transaktion kann es durchaus zu kurzzeitigen Inkonsistenzen kommen.

Isolation

Die Isolation ist eine wichtige Eigenschaft für verteilte Systeme. Bei Mehrbenutzerbetrieb muss gewährleistet sein, dass Anfragen eines Benutzers isoliert von anderen behandelt werden. Es darf zu keiner Überschreibung oder Änderung von Daten der anderen Benutzer kommen. Dafür können verschiedene Synchronisationsmaßnahmen genutzt werden [HR99].

Dauerhaftigkeit

Wurde eine Transaktion einmal erfolgreich ausgeführt, müssen ihre Änderungen dauerhaft gespeichert sein. Durch etwaige Hardwarefehler oder Systemabstürze dürfen die Änderungen nicht verloren gehen [HR99]. Auch hierzu wird, wie bei den Eigenschaften *Atomarität* und *Konsistenzerhaltung* das Logging verwendet. Allerdings nicht um Änderungen rückgängig zu machen, sondern um im Fehlerfall Transaktionen erneut ausführen zu können.

3.2.2 BASE

Das Gegenstück zu ACID ist das sogenannte BASE[2]-Prinzip. Innerhalb eines verteilten Systems ist in diesem Kontext, *Konsistenz* erreicht, wenn alle Kopien eines Datums identisch sind. Sind alle Knoten **immer** auf demselben Stand, ist die *strikte Konsistenz* erreicht. Bei BASE geht man von einer *schließlichen Konsistenz* aus. Diese besagt,

[2] Basically Available, Soft state, Eventual consistency; dt. grundsätzlich verfügbar, unsicherer Zustand, schließliche Konsistenz

dass ein Datensatz nicht sofort über alle Knoten konsistent sein muss, sondern dass dies längere Zeit dauern kann. Der Client muss hierbei das Lesen von älteren Kopien verarbeiten können.

Hierfür sind mehrere Konzepte definiert, die sich zwischen *strikter Konsistenz* und *schließlicher Konsistenz* bewegen. Diese unterteilt man in *client-centric consistency*, die Konsistenz auf Clientseite und *data-centric consistency* [Vog07], die Konsistenz innerhalb der Daten.

3.2.3 Einordnung in das CAP-Theorem

Auf Basis der Kapitel können ACID und BASE nun im Kontext von verteilten System einfach in das CAP-Theorem einordnet werden. Dabei ist der Hauptunterschied, wie in den Kapiteln 3.2.1 und 3.2.2 beschrieben, die Konsistenz. BASE hat eine niedrige Anforderung an die Konsistenz, dagegen ist die Verfügbarkeit sehr wichtig. BASE kann, wie in Abbildung 3.3 gezeigt, in den Bereich *Partitionstoleranz - Verfügbarkeit* eingeordnet werden. Da ACID strikte Anforderungen an die Konsistenz hat, wird es der *Konsistenz - Partitionstoleranz* zugeordnet.

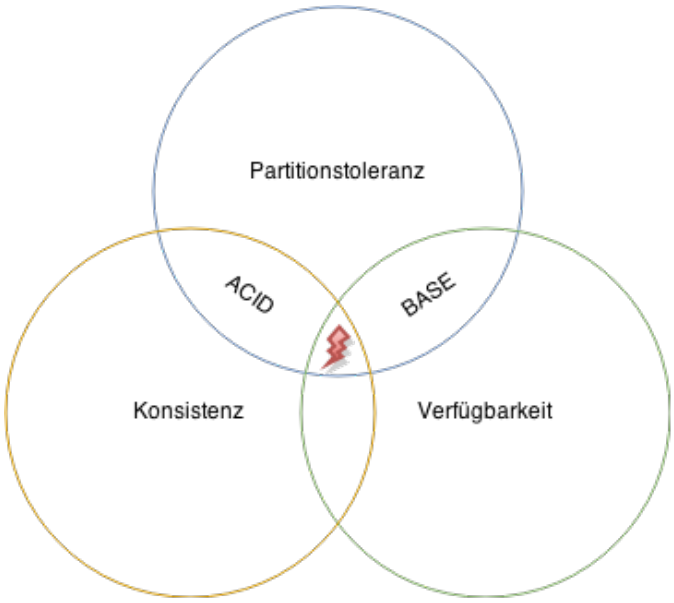


Abbildung 3.3: ACID und BASE innerhalb des CAP-Theorems

3.3 Grundlegende Mechanismen verteilter Systeme

Für die Realisierung der im Kapitel 3.2 genannten Konzepte gibt es einige einfache Mechanismen. Da diese für die Konzeption (siehe Kapitel 5) und spätere Implementierung (siehe Kapitel 6) wichtig sind, werden diese kurz vorgestellt und diskutiert. Dabei wird, wenn möglich, ein Beispiel für eine real existierende Implementierung angegeben.

3.3.1 Replikation

Um Daten gegen den Ausfall von Hardware abzusichern, können diese auf mehreren Maschinen an verschiedenen Standorten gespeichert werden. Dabei werden die Daten von einem Standort zu einem anderen *repliziert*. Fällt nun die erste Datenquelle aus, kann die zweite replizierte Quelle die Aufgaben der ersten ohne Ausfall des Dienstes übernehmen. Die Konsistenz der Daten bleibt dabei erhalten [MDM13]. Es kann auch in beide Richtungen repliziert werden, um Änderungen an beiden Datenbeständen möglich zu machen. Der Nachteil der Replikation besteht darin, dass die Daten vollständig synchronisiert werden müssen, auch wenn möglicherweise nur ein Teil benötigt wird. Dies führt im Umkehrschluss zu einer niedrigeren Verfügbarkeit. Viele Datenbanken bringen heute eingebaute Funktionen zur Replikation von Haus aus mit.

3.3.2 Versionierung

Mithilfe einer *Versionierung* ist es möglich, Änderungen innerhalb eines Datensatzes nachzuvollziehen. Dabei bekommt jeder Datensatz eine Versionsnummer zugewiesen, die bei jedem Schreibzugriff inkrementiert wird. Dadurch ist es für einen Client möglich zu prüfen, ob sein Datensatz noch aktuell ist oder in der Zwischenzeit Änderungen durchgeführt wurden. Ist dies der Fall, so müssen die Daten aktualisiert werden, bevor Änderungen durchgeführt werden können. Die *Versionierung* kann mit der *Replikation* kombiniert werden, um die zu replizierende Datenmenge zu verringern.

Als Beispiel für eine Implementierung kann hier das Codeverwaltungstool *GIT* genannt werden [Cha09]. Bei *GIT* wird jeder eingetragenen Änderung eine eindeutige Versi-

onsnummer zugewiesen. Dadurch kann nachvollzogen werden, in welcher Reihenfolge Änderungen durchgeführt wurden.

3.3.3 Caching

Beim *Caching* [ALK+02] werden einmal abgerufene Daten für einen gewissen Zeitraum zwischengespeichert. Dadurch wird der wiederholende lesende Zugriff auf Daten erheblich beschleunigt. Außerdem kann auf Daten, die gerade durch eine Transaktion gesperrt sind, trotzdem lesend zugegriffen werden. Werden Daten geschrieben, müssen diese allerdings an die entfernte Anwendung weitergegeben werden, um Inkonsistenz zu vermeiden. Außerdem müssen alte Versionen im Cache regelmäßig aktualisiert werden. Kombiniert man die beiden vorhergehenden Konzepte mit dem neu eingeführten *Caching*, so ergeben sich einige Vorteile. Zum einen wird die zu übertragende Datenmenge reduziert, zum anderen kann die Anzahl der Änderungen an Datensätzen nachvollzogen werden. Ein Beispiel für den Einsatz von Caching sind Webproxys wie squid [WC98].

3.3.4 Sperrfunktionen

Bei *Sperrfunktionen* werden Teile der Daten in einer Datenbank für die Dauer einer Transaktion gesperrt. Das heißt, dass in diesem Zeitraum der Zugriff auf die Daten nur eingeschränkt, zum Beispiel nur lesend, oder gar nicht möglich ist. Es stehen dabei typischerweise Lese- und Schreibsperren mit verschiedener Granularität zur Verfügung. Dies ermöglicht es, nur einzelne Datensätze zu sperren, anstatt einer ganzen Datenbank [GR09]. Grundsätzlich gibt es zwei unterschiedliche Möglichkeiten, Sperrfunktionen in Datenbanken zu implementieren.

Optimistic Locking

Optimistic Locking ist vor allem für Systeme mit vielen, parallelen Lesezugriffen und wenigen überlappenden Schreibzugriffen geeignet. Zugriffskonflikte werden minimiert, da diese nur bei Schreibzugriffen auftreten können [Kun79]. Die eigentliche Sperre wird

erst gesetzt, wenn die Daten wirklich geändert werden. Dabei wird nach dem System *First come, first served.* vorgegangen. Die Transaktion, die zuerst eine Änderung an die Datenbank überträgt, hat das Recht Änderungen durchzuführen. Andere Transaktionen müssen warten, bis die erste Transaktion beendet ist.

Pessimistic Locking

Im Gegensatz zu *Optimistic Locking* werden beim *Pessimistic Locking* zu Beginn der Transaktion alle Lese- und Schreibsperren gesetzt. Für andere Transaktionen der Datenbank ist in diesem Zeitraum kein Zugriff möglich. Dadurch kann es zu langen Wartezeiten für andere Prozesse kommen. Diese können zum Abbruch der Transaktion führen [ENS02]. Vorteile hat dies vor allem bei Anwendungen, die für einen Großteil der Anfragen Änderungen an den Daten durchführen.

3.3.5 Transaction Log

Beim Verfahren *Transaction Log* werden alle Änderungsoperationen am Datenbestand in ein separates Logfile geschrieben. Dies ermöglicht, dass bei einem Absturz oder Fehlern die Daten wiederhergestellt werden. Außerdem können alle Änderungen nachvollzogen werden und Clients über Veränderungen in ihrem Datenbestand informiert werden. Der Nachteil besteht darin, dass das Logfile sehr groß werden kann. Dieses Konzept wird in vielen Anwendungen in Abwandlungen verwendet. Ein Beispiel ist das Codeverwaltungs-Tool GIT, bei dem eine Form von Transaction Log dazu verwendet wird, die Änderungen im Sourcecode zu dokumentieren und eine Versionsverwaltung möglich machen. Ein anderes Beispiel ist das Dateisystem ZFS [BAH+03]. Hier werden Änderungen im Dateisystem protokolliert, um die Konsistenz der Daten zu gewährleisten und im Falle eines Defekts einer Festplatte die Daten wiederherstellen zu können.

4

Qualitätskriterien und Anforderungen

Die Anforderungen an ein Konzept zur Synchronisation und Fehlertoleranz für den proCollab-Prototypen sollen nun präsentiert und diskutiert werden. Zunächst werden hierfür repräsentative Anwendungsfälle erstellt, die dann später in konkretere Anforderungen überführt werden. Außerdem werden in diesem Abschnitt allgemeine Problemstellen betrachtet, die bei verteilten Systemen auftreten. Mit dem Ziel der Verbesserung der Synchronisation der Daten zwischen pCC und dem pCP liegt das Hauptaugenmerk bei den Anforderungen und Problemen der Synchronisation.

4.1 Allgemeine Herausforderungen verteilter Client-Server-Systeme

Es gibt einige allgemeine Herausforderungen bei verteilten Client-Server-Systemen. Diese werden im Folgenden beschrieben, um die Problematik zu verdeutlichen.

4.1.1 Verfügbarkeit des Dienstes

Unter Verfügbarkeit versteht man ein Maß nach dem ein System bestimmte Anforderungen nach einer bestimmten Zeit erfüllt. Im Kontext des pCC bedeutet dies, dass Anfragen an den pCP in einer bestimmten Zeit beantwortet werden. Die *Verfügbarkeit* eines Dienstes für einen mobilen Client beeinflusst maßgeblich die Benutzbarkeit des Systems. Diese wird von mehreren Faktoren beeinflusst [Tv08]: So kann zum einen (siehe Abbildung 4.1) die Verbindung zwischen Server und Netzwerk (z.B. Internet) gestört sein. Andererseits kann auch die Verbindung zwischen Client und Internet gestört oder ganz unterbrochen sein (siehe Abbildungen 4.2 und 4.3). Ein weiterer Faktor, der die Verfügbarkeit beeinflusst, ist die aktuelle Auslastung der eigentlichen Serverhardware. Im Kontext des pCP muss dies bei der Kommunikation mit dem pCC beachtet werden. Im speziellen darf bei unvollständiger Datenübertragung aufgrund einer abgebrochenen oder verzögerten Netzwerkverbindung keine Änderungen durchgeführt werden.

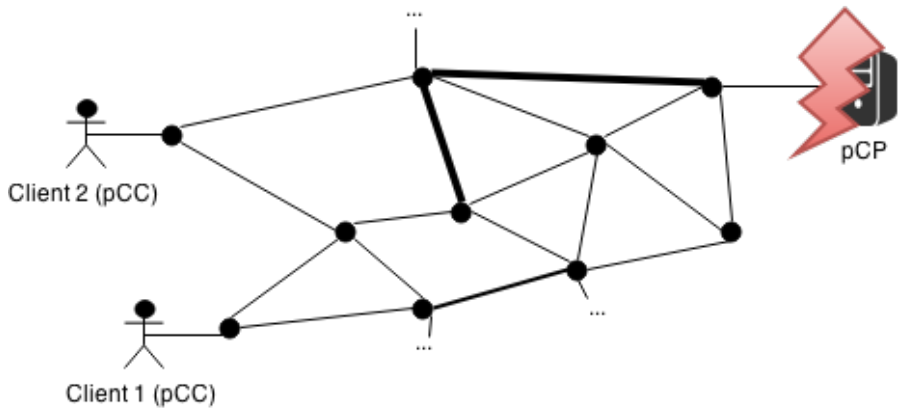


Abbildung 4.1: Verfügbarkeit des Dienstes

4.1.2 Latenz der Verbindung

Latenz beschreibt die Zeit zwischen Auslösung einer Anfrage (Request) bis zum erhalten einer Antwort (Response). Diese wird in mobilen Applikationen von mehreren Faktoren beeinflusst [Tv07]. So kann die Verbindung zwischen Client und Netzwerk gestört oder unterbrochen sein. Zum anderen können diese Verbindungsprobleme auch außerhalb des Einflussbereichs des Clients liegen (siehe Abbildung 4.2). Es ist also nicht möglich, die Latenz im Voraus genau zu bestimmen. Anwendungen sollten deswegen immer mit Hinblick auf variable Antwortzeiten und niedrige Verbindungsgeschwindigkeiten optimiert werden.

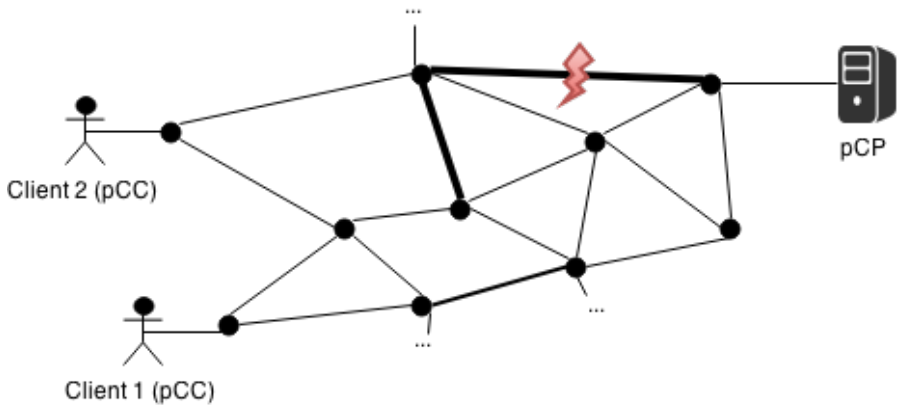


Abbildung 4.2: Latenz der Verbindung

4.1.3 Inkonsistente Internetverbindung

Eine weitere Schwierigkeit ist die inkonsistente Internetverbindung im mobilen Datennetz (siehe Abbildung 4.3). Durch Schwächen in der Netzabdeckung und Empfangsproblemen durch sich bewegende Benutzer ist eine durchgehende Internetverbindung für den pCC nicht garantiert. Dadurch kann es leicht zum Verlust von Datenpaketen kommen [Cou12]. Die Anfragen können in diesem Fall vom pCP nicht beantwortet werden. Unvollständige Änderungen werden deshalb abgewiesen. Kommt es bei der Antwort des pCP an den pCC für den Client zu Unterbrechungen darf die Änderung auf dem pCC nicht

eingetragen werden. In beiden Fällen muss der pCC seine Anfrage ein weiteres Mal ausführen.

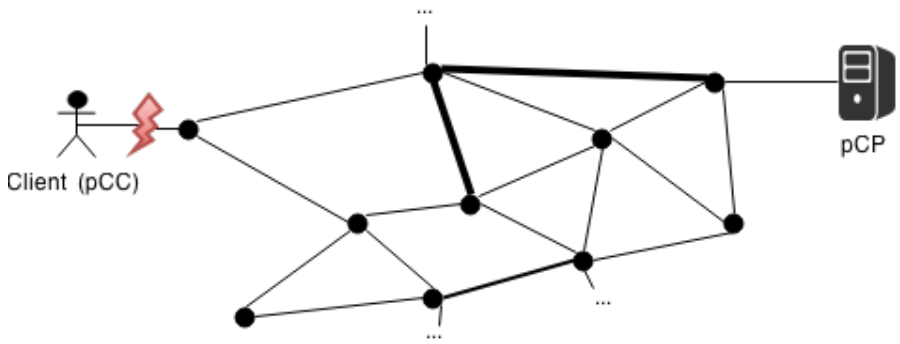


Abbildung 4.3: Inkonsistente Internetverbindung

4.1.4 Uhrenvergleich

Unter *Uhrenvergleich* versteht man, dass Daten und Aktionen der Benutzer von der Zeit abhängig sind. Dazu verfügen die Daten über einen Zeitstempel, der bei jeder Aktion mit übertragen werden muss. Aktionen, die mit einem veralteten Zeitstempel ausgeführt werden, können dabei unter Umständen abgewiesen werden. Dadurch, dass der Server immer über die aktuellste Version innerhalb des Systems verfügt, soll zu jedem Zeitpunkt die Validität der Daten garantiert sein. In Abbildung 4.4 wird ein möglicher Konflikt gezeigt. Bob verfügt nicht über die korrekte Uhrzeit und kann deshalb den Datensatz nicht editieren.

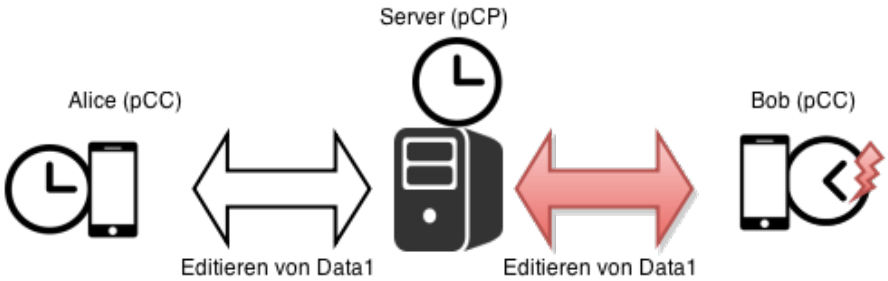


Abbildung 4.4: Zeitproblem

4.1.5 Parallele konkurrierenden Zugriffe

Bei parallelen konkurrierenden Zugriffen gibt es einige bekannte Probleme. Im Folgenden werden diese diskutiert und mögliche Lösungswege besprochen.

Verlorenes Update

Zu einem verlorenen Update kommt es wenn zwei parallel laufende Transaktionen den selben Datensatz bearbeiten. Dabei wird die Änderung der ersten Transaktion direkt von der zweiten überschrieben [KE11]. Eine Möglichkeit diesen Fehler zu verhindern, sind die in Kapitel 3.3.4 vorgestellten Sperrfunktionen.

Phantomproblem

Das Phantomproblem tritt auf, wenn eine Transaktion eine Menge von Datensätzen mit bestimmten Eigenschaften liest und eine weitere Transaktion zum selben Zeitpunkt weitere Elemente zu dieser Menge hinzufügt. Konkret bedeutet dies, dass die Menge der zurückgegeben Datensätzen nicht vollständig und damit nicht valide ist. Die einfachste Möglichkeit dies zu verhindern ist es die komplette Tabelle für die Zeit der Abfrage zu sperren (Kapitel 3.3.4). Eine weitere Option ist das Einführen von eindeutigen Versionsnummern auf Tabellenebene (Kapitel 3.3.2). Dadurch kann dem Client eine veraltete aber valide Menge an Datensätzen zurückgegeben werden [ENS02].

Schreib-Lese-Konflikt

Ein Schreib-Lese-Konflikt tritt auf, wenn eine Transaktion einen Datensatz liest, den eine andere Transaktion zum selben Zeitpunk bearbeitet, diesen Vorgang aber noch nicht abgeschlossen hat [HR99]. In diesem Fall können an die lesende Transaktion fehlerhafte Datensätze zurückgegeben werden. Zur Verhinderung des Schreib-Lese-Konflikts werden Sperrfunktionen verwendet.

Nichtwiederholbares Lesen

Die Problematik des nichtwiederholbaren Lesens tritt auf, wenn innerhalb einer Transaktion das wiederholte Lesen desselben Datensatzes unterschiedliche Ergebnisse zurück liefert, weil der Datensatz in der Zwischenzeit von einer weiteren Transaktion bearbeitet wurde. Im Gegensatz zum Schreib-Lese-Konflikt ist die zweite Transaktion bereits beendet. Auch dieses Problem kann durch Sperrfunktionen verhindert werden [HR99].

4.2 Anwendungsfälle

In diesem Kapitel werden Anwendungsfälle vorgestellt, die verschiedene Szenarien der Client-Server-Kommunikation betrachten. Das Hauptaugenmerk liegt dabei auf Fällen die Konflikte bei der Synchronisation auslösen können. Dabei sollen Probleme erkannt werden, die beim Mehrbenutzerbetrieb mit verteilten Clients auftreten können, um diese im späteren Konzept adressieren zu können.

Um die Anwendungsfälle zu illustrieren, werden im Folgenden die beiden Benutzer *Alice* und *Bob* verwendet. Diese kommunizieren mit dem pCP über den pCC (Abbildung 4.5). Zur Veranschaulichung löst Bob dabei immer die erste Anfrage aus und Alice die zweite.

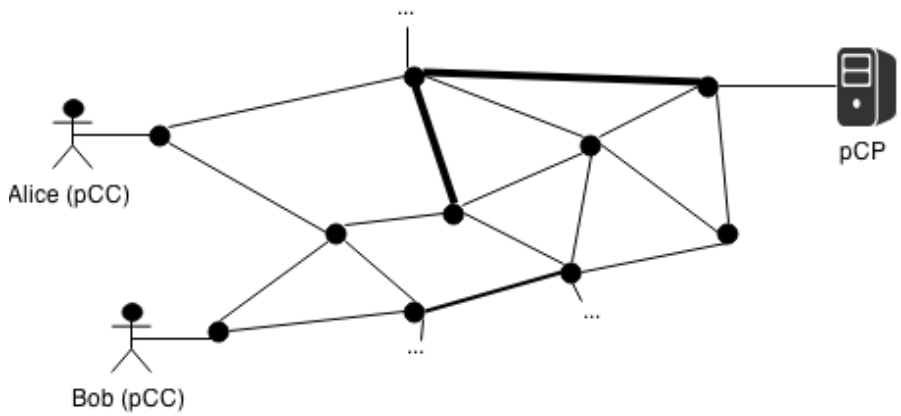


Abbildung 4.5: Alice and Bob

4.2.1 Anwendungsfälle: Benutzer

Hier werden die Anwendungsfälle aus Sicht des Benutzers (siehe Kapitel 2.6.4) behandelt.

AF1: Bearbeitung von verschieden Attributen eines Eintrags

Beschreibung: Alice und Bob bearbeiten den gleichen Eintrag einer Checkliste. Bob bearbeitet das Attribut *name* und Alice das Attribut *text*.

Durchführung: AF1 läuft im Bezug auf die parallelen konkurrierenden Zugriffe ohne Konflikte ab. Da Alice und Bob verschiedene Attribute des Eintrags bearbeiten, können die Änderungen innerhalb des gleichen Eintrags ohne Fehler zusammengeführt werden. In Abbildung 4.6 wird der Ablauf zum besseren Verständnis anhand eines Beispiels gezeigt. In diesem Schaubild ist der Ausgangszustand des Eintrags auf dem pCP mit *V1* markiert. Der Endzustand des Eintrags ist mit *V2* markiert.

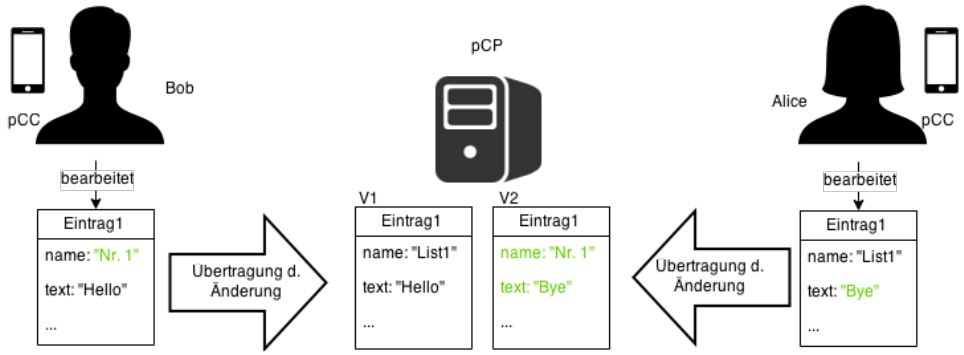


Abbildung 4.6: AF1: Änderungen zusammenführen

AF2: Bearbeitung von gleichen Attributen eines Eintrags

Beschreibung: Alice und Bob ändern beide den Namen eines Eintrags ohne zuvor die aktuelle Version des pCP synchronisiert und damit die Änderungen des anderen erhalten zu haben.

Durchführung: Bei AF2 treten mit Hinblick auf den parallelen Zugriff Konflikte auf. Die Änderung von Bob auf dem pCP wird von Alice sofort überschrieben. Dies ist ein klassischer Fall von *Lost Update* (siehe Kapitel 4.1.5). Nach Abschluss beider Änderungsvorgänge ist die Änderung von Bob verloren gegangen (Abbildung 4.7).

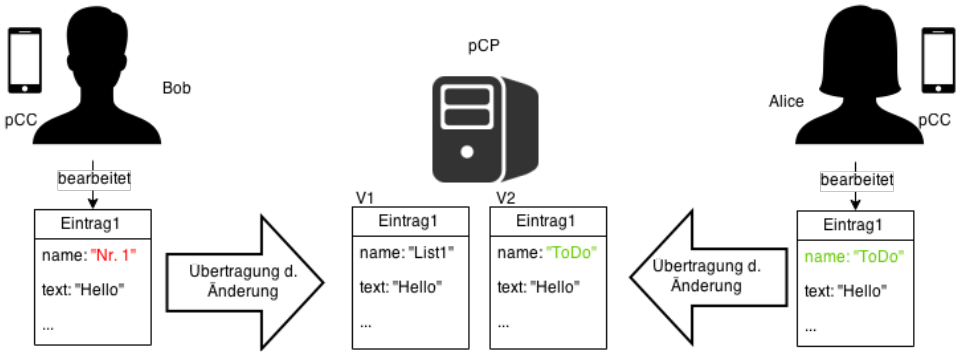


Abbildung 4.7: AF2: Eintrag nach Änderungen mit Konflikten.

AF3: Einfügen von Elementen I

Beschreibung: Bob fragt alle Einträge einer Checkliste ab. Alice fügt gleichzeitig einen neuen Eintrag hinzu.

Durchführung: Bei AF3 kommt es aufgrund der parallelen Zugriffe zu Konflikten. Konkreter ist die Rückgabe, die Bob erhält, zu diesem Zeitpunkt schon nicht mehr aktuell. Der von Alice hinzugefügte Eintrag ist nicht vorhanden (Abbildung 4.8). Dieses Verhalten ist als Phantomproblem bekannt (siehe Kapitel 4.1.5).

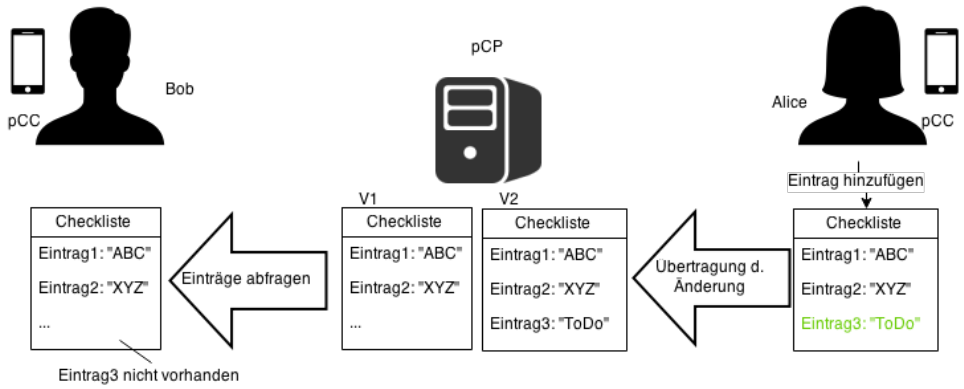


Abbildung 4.8: AF3: Phantomproblem

AF4: Einfügen von Elementen II

Beschreibung: Alice und Bob fügen parallel neue Einträge an beliebiger Stelle in die Checkliste ein.

Durchführung: Bei AF4 treten Konflikte auf. Hat Bob sein Element an einer beliebigen Stelle eingefügt, ändert sich der Kontext der Reihenfolge für Alice. Fügt Alice nun an einer beliebigen anderen Stelle einen Eintrag ein, stimmt die Reihenfolge innerhalb der Liste nicht mehr. Dadurch können Einträge überschrieben werden (siehe Abbildung 4.9)

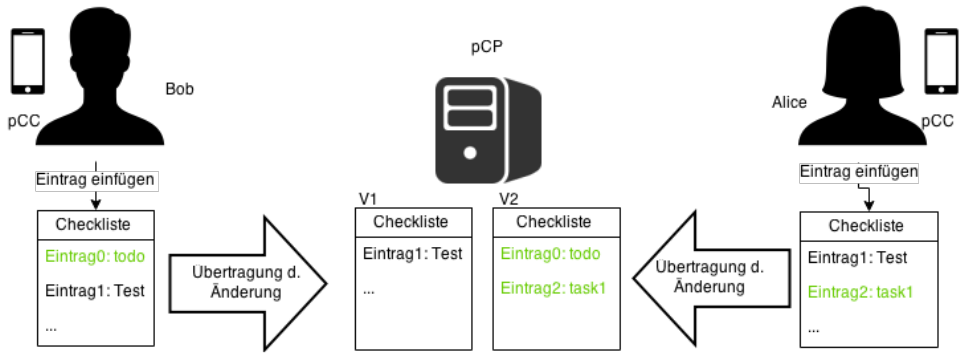


Abbildung 4.9: AF4: Einträge an beliebiger Stelle in Checkliste einfügen.

AF5: Löschen und Bearbeiten von Einträgen I

Beschreibung: Bob löscht Eintrag3 aus der Checkliste. Alice bearbeitet den Namen von Eintrag1 in derselben Checkliste.

Durchführung: Dieser Anwendungsfall läuft ohne Konflikte ab. Alice und Bob arbeiten auf verschiedenen Teilen der Checkliste. Die Änderungen der beiden können zusammengeführt werden, weil sich die Checkliste nicht ändert. Es werden ausschließlich die beiden eindeutige identifizierbaren Einträge bearbeitet.

AF6: Löschen und Bearbeiten von Einträgen II

Beschreibung: Bob löscht Eintrag3 aus der Checkliste. Alice fügt einen neuen Eintrag am Ende der Checkliste ein.

Durchführung: AF6 läuft ohne Probleme ab. Wenn Bob Eintrag3 löscht, ändert sich zwar die Reihenfolge innerhalb der Checkliste. Das Ende der Checkliste ist aber ein Spezialfall, da dieses immer eindeutig bestimmbar ist. Das Element kann ohne Konflikte am Ende der Checkliste angehängt werden.

AF7: Löschen und Bearbeiten von Einträgen III

Beschreibung: Bob entfernt das erste Element aus der Checkliste. Alice fügt ein neues Element an beliebiger Position der Checkliste ein.

Durchführung: In AF7 gibt es durch die parallelen Zugriffe einen Konflikt. Nachdem Bob den Eintrag gelöscht hat, wurde die Reihenfolge in der Checkliste verändert. Der pCC bei Alice muss zunächst die neuste Version der Checkliste holen. Ihre Änderungen werden nicht eingetragen (Abbildung 4.10).

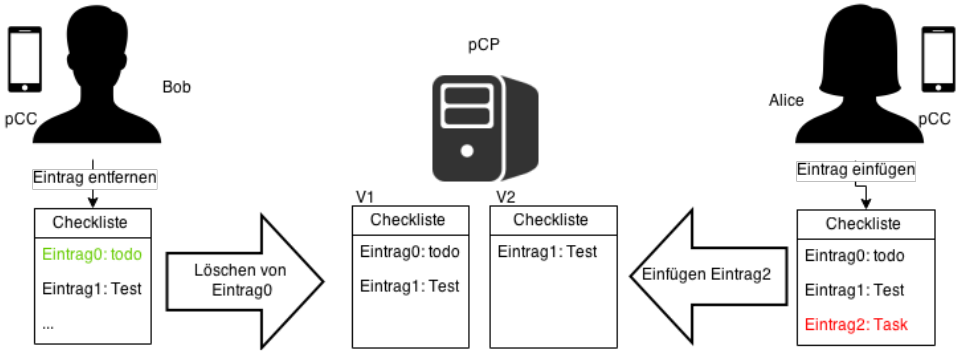


Abbildung 4.10: AF7: Elemente löschen und hinzufügen

AF8: Verschieben und Bearbeiten von Elementen

Beschreibung: Bob verschiebt einen Eintrag innerhalb der Checkliste und Alice bearbeitet einen anderen Eintrag in der Checkliste.

Durchführung: In AF8 gibt es keine Konflikte. Alice und Bob bearbeiten verschiedene Teile der Checkliste, die eindeutig identifizierbar sind. Alice bearbeitet den Inhalt eines **Eintrags**, während Bob einen Eintrag innerhalb der **Checkliste** verschiebt (Abbildung 4.11). Diese beiden Elemente können gleichzeitig bearbeitet werden.

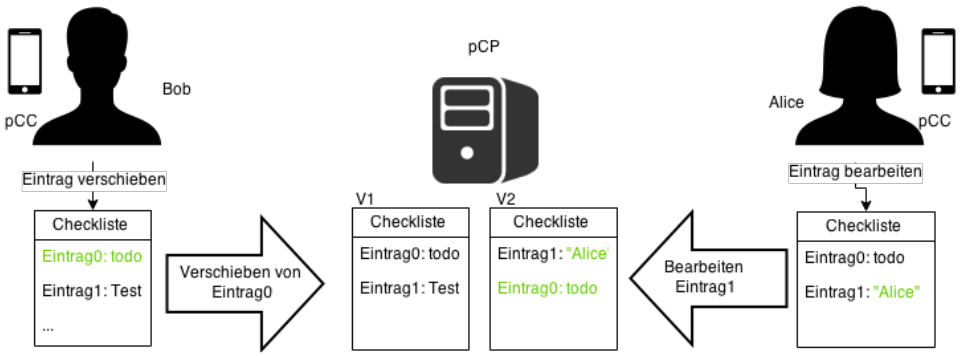


Abbildung 4.11: AF8: Elemente verschieben

AF9: Statusänderung

Beschreibung: Der Status eines Eintrags wurde auf *abgeschlossen* gesetzt. Nun versucht Bob den Namen dieses Eintrags zu verändern.

Durchführung: Bei AF9 tritt ein Konflikt auf. Ist ein Eintrag einmal auf den Status *FINISHED* gesetzt, sind keine Änderungen mehr möglich. Die Änderung wird abgelehnt (Abbildung 4.12)

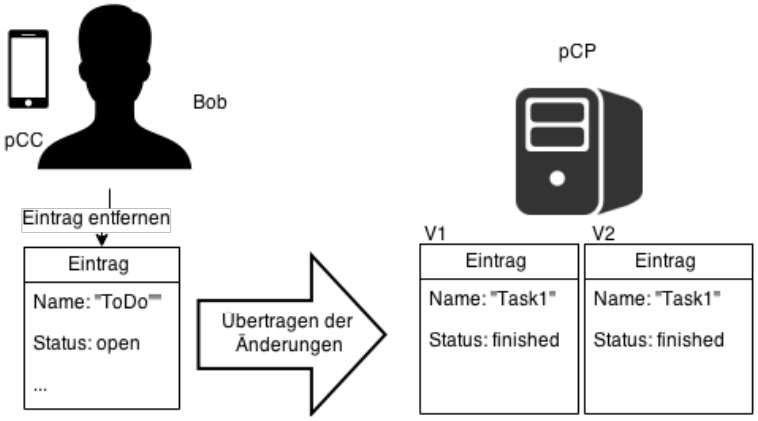


Abbildung 4.12: Anwendungsfall: Element verändern mit falschem Status

4.2.2 Verwalter

Nachdem die Anwendungsfälle des Benutzers vorgestellt wurden, werden nun die des Verwalters betrachtet. Dabei ist festzuhalten, dass dieselben Anwendungsfälle, die für den Benutzer gefunden wurden, auch für den Verwalter gelten. Er hat prinzipiell dieselben Anwendungsfälle wie der in Kapitel 4.2.1 vorgestellte Benutzer.

AF10: Alice und Bob bearbeiten die beteiligten Personen einer Checklisteninstanz.

Beschreibung: Alice und Bob sind beide Verwalter derselben Liste. Nun wollen beide an den beteiligten Personen einer Checkliste Änderungen vornehmen.

Durchführung: In AF10 können Konflikte auftreten, die auch beim Bearbeiten von normalen Elementen auftreten. Es hängt von der Checkliste und der Aktion ab, ob in diesem Fall Konflikte auftreten.

4.3 Analyse der Anwendungsfälle

Durch diese Anwendungsfälle wurden die klassischen Probleme aufgezeigt, die bei einem verteilten System auftreten können. Typischerweise treten Konflikte auf, wenn mehrere Clients Änderungen an den gleichen Einträgen zur selben Zeit vornehmen. So kann es sein, dass ein Client die neueste Version eines Eintrags noch nicht zur Verfügung hat und deshalb auf der Grundlage veralteter Annahmen arbeitet. Geht man nun von den klassischen Methoden ’erstellen’, ’lesen’, ’aktualisieren’ und ’löschen’ zum Bearbeiten der Entitäten auf Serverseite aus, so müssen für die Aktionen ’erstellen’, ’aktualisieren’ und ’löschen’ Regeln geschaffen werden, um zu garantieren, dass keine Änderungen durch die parallelen konkurrierende Zugriffe verloren gehen.

4.3.1 Ableitung der Anforderungen

Aus den oben aufgeführten Anwendungsfällen lassen sich nun konkrete Anforderungen ableiten.

Konsistenz

Das verteilte System soll die Eigenschaft *Konsistenz* erfüllen. Das heißt, die Daten auf dem Client und dem Server müssen sich nach einer Datenübertragung in dem selben

Zustand befinden. Im Speziellen bedeutet dies das auf dem pCC und pCP die gleiche Version vorliegt. Dabei können übertragene Änderungen durch Zusammenführung mit anderen Änderungen weiter verändert werden. Bei nicht lösbaren Konflikten sollen die Daten auf dem Client nicht verworfen werden, sondern es sollen Möglichkeiten zur Konfliktlösung angeboten werden.

Fehlertoleranz

Das verteilte System muss *fehlertolerant* sein. Bei Fehlern in der Datenübertragung oder Verarbeitung müssen Fehlerfälle abgefangen und das System in einen validen Zustand übertragen werden. Im Speziellen bedeutet dies, dass bei nicht lösbaren Konflikten und fehlerhafter Datenübertragung keine Daten verloren gehen. Über Fehler muss der Benutzer informiert werden.

Verteilung der Daten

Die Daten sollen innerhalb des verteilten Systems verteilt werden. Für den Benutzer relevante Daten sollen auf dem Client zwischengespeichert werden. Sind die Daten veraltet, werden diese automatisch aktualisiert. Werden Änderungen auf dem Client durchgeführt, sollen diese möglichst direkt an den Server übertragen werden. Ist dies aufgrund fehlender Internetverbindung nicht möglich, muss dies dem Benutzer angezeigt werden und die Änderungsmöglichkeiten eingeschränkt werden, um Konflikte zu vermeiden.

Einordnung in CAP

Nach den Anforderungen sollen die pCCs Daten (z.B. Checklisten) zwischenspeichern und nach Möglichkeit an den pCP übertragen. Außerdem soll die Konsistenz der Daten immer erhalten bleiben. Dabei ist zum einen die Konsistenz innerhalb der Daten gemeint, zum anderen, dass alle pCCs mit den gleichen Daten arbeiten. Es ist in dem Fall des pCP wichtiger, dass die Daten immer konsistent bleiben, deshalb werden kleine

Einschränkungen bei der Verfügbarkeit in Kauf genommen. Diese beschränken sich darauf, dass es bei einzelnen Änderungen von Clients zu Konflikten kommen kann, die eventuell manuell gelöst werden müssen. Auf Basis der Grundlagen folgt daraus, dass das Konzept von *proCollab* auf Basis des ACID-Prinzips erstellt werden sollte (Abbildung 4.13).

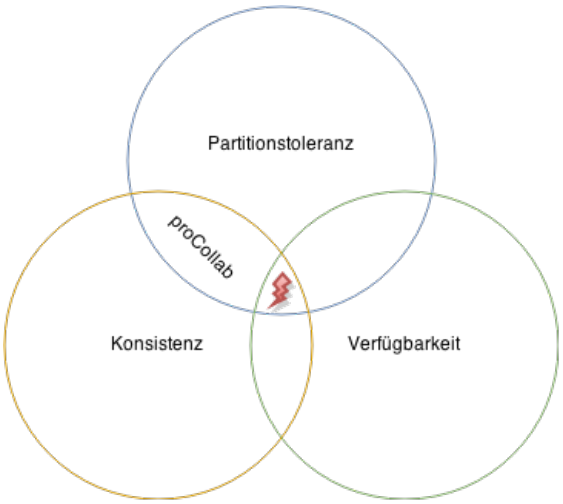


Abbildung 4.13: CAP-Theorem: proCollab

Folgende Abbildung 4.14 fasst die nötigen Anforderungen die der pCP erfüllen muss zusammen.

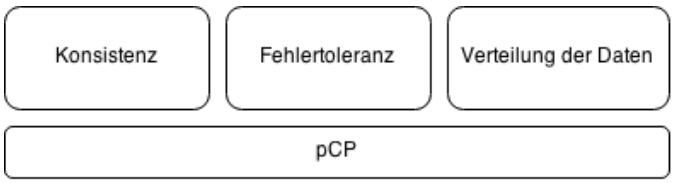


Abbildung 4.14: Zusammensetzung der Anforderungen

5

Konzept zur Synchronisation und Fehlertoleranz

Auf Basis der Kapitel 2, 3 und 4, in denen die dafür notwendigen Grundlagen und Anforderungen vorgestellt wurden, wird nun das Konzept für die Synchronisation und Fehlertoleranz für den proCollab-Prototyp diskutiert. Dazu wird das Augenmerk zunächst auf die aktuelle Architektur und die Analyse der Anwendungsfälle aus dem vorherigen Kapitel gelegt (siehe Abbildung 2.1).

In den Grundlagen wurden Mechanismen, wie z.B. die Sperrfunktionen (siehe Kapitel 3.3.4 beschrieben, die helfen, die Konsistenz von verteilten Daten zu erhalten. Im Speziellen werden die Mechanismen Changelog, Versionierung, Sperrfunktionen und Caching eingesetzt. Dabei wird zwischen dem Konzept für den pCC und pCP unterschieden, da hier jeweils verschiedene Ansätze nötig sind. In Abbildung 5.1 ist die Verteilung der

Mechanismen illustriert. Konkret wird gezeigt, welche Mechanismen auf dem pCC und welche auf dem pCP implementiert werden müssen.

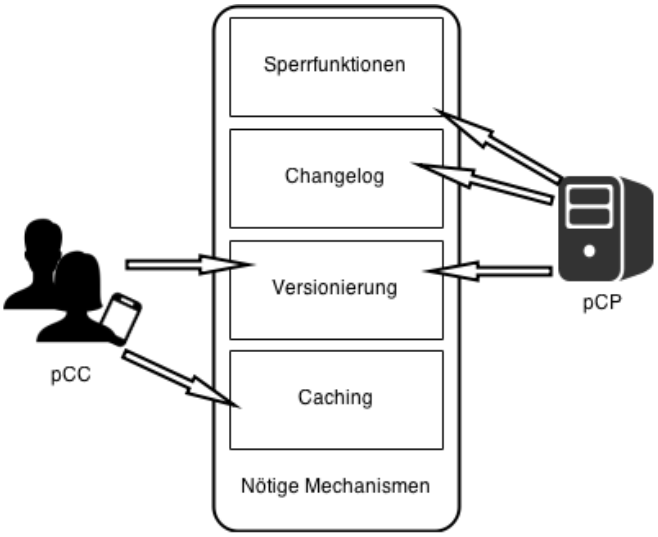


Abbildung 5.1: Benötigte Mechanismen

5.1 Verwendete Mechanismen zur Fehlervermeidung

Caching

Auf Seite des pCC ist es nötig, Caching einzuführen. Dadurch soll garantiert werden, dass das System auch bei nicht vorhandener Internetverbindung benutzbar bleibt. Auf Seite des pCP ist dies nicht notwendig, da im Regelfall eine stabile Internetverbindung besteht und der pCP zu jeder Zeit nur valide Daten vorhält.
Bei nicht vorhandener Internetverbindung sollen Änderungen auf dem pCC zwar lokal eingetragen werden, gleichzeitig muss dem Benutzer aber mitgeteilt werden, dass seine lokalen Änderungen noch nicht auf dem pCP eingetragen sind. Dies ist deshalb wichtig, weil es bei der Übertragung zum pCP zu Konflikten kommen kann (siehe Kapitel 4). Erst wenn die Änderungen erfolgreich auf den pCP übertragen wurden, das heißt ohne Konflikte oder mit aufgelösten Konflikten eingetragen werden konnten, dürfen diese auf dem pCC als aktualisiert markiert werden. Hier muss der Benutzer ebenfalls

Rückmeldung bekommen.
Ist eine Internetverbindung vorhanden, soll der Cache möglichst transparent agieren, d.h. Änderungen sollen nur für kurze Zeit im Cache gespeichert werden. In Abbildung 5.2 (a) ist der Fall der nicht vorhandenen Internetverbindung dargestellt. Und in Abbildung 5.2 (b), der des transparenten Caches.

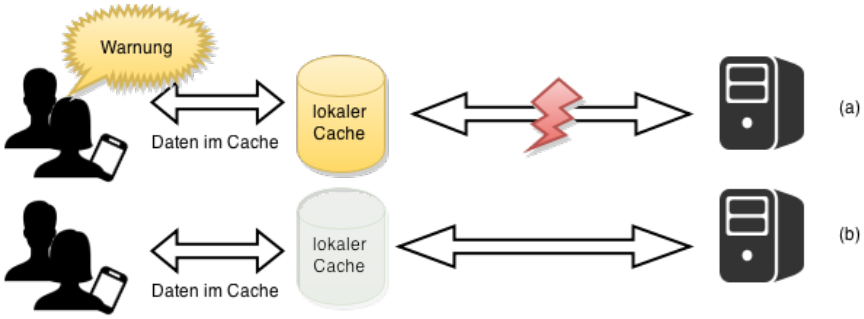


Abbildung 5.2: proCollab-Client: Cache auf der Clientseite

Versionierung

Die Versionierung der Datensätze muss sowohl auf der Seite des pCC, als auch auf der Seite des pCP unterstützt werden. Nachdem die Versionierung eingeführt wurde, erhält jeder Datensatz, wenn er aktualisiert, das heißt schreibend auf ihn zugegriffen wird, eine Versionsnummer. Durch diese Versionsnummer ist es anschließend möglich, verschiedene Versionen des gleichen Datensatzes zu unterscheiden. Im Speziellen soll es möglich sein, einen Datensatz wie zum Beispiel einen Eintrag, zu jeder Zeit eindeutig identifizieren zu können. Ist diese eindeutige Identifizierung möglich und können verschiedene Versionen desselben Datensatzes in Beziehung zueinander gesetzt werden, dadurch ist eine Konfliktlösung möglich. Wird nun eine Änderung an einer älteren Version eines Datensatzes an den pCP übertragen, kann dieser versuchen die Änderung auf Basis der alten Versionen einzutragen und auf die aktuelle Version des Datensatzes zu übertragen.

Changelog

Die vorletzte Komponente im Konzept zur Synchronisation und Fehlertoleranz ist die Einführung eines Changelogs (siehe Kapitel 3.3.5). Dies wird nur auf dem pCP eingesetzt, um erfolgreiche Änderungen zu dokumentieren und nachvollziehen zu können. Dadurch ist es unter anderem möglich, Konflikte, die bei Änderungen auftreten können, teilweise automatisch zu lösen (siehe Kapitel 5.1.1). Dazu wird für jede Änderung eine Zeile in das Changelog geschrieben. Diese enthält die Änderung und die jeweils zu diesem Zeitpunkt aktuelle Versions-ID. Der pCP hat dadurch die Möglichkeit, bei einer Änderung mit einer bestimmten Versions-ID, ausgehend von dieser, alle Änderungen nachzuvollziehen und zu versuchen, die veralteten Änderungen mit den aktuellen zusammenzuführen. Zum anderen werden auf Basis dieser gespeicherten Änderungen jeweils nur die benötigten Deltas der Änderungen an die pCCs übertragen.

Sperrfunktionen

Sperrfunktionen sollten auf dem pCP realisiert werden und dabei das Verfahren Optimistic Locking verwenden. Da durch den Einsatz von Optimistic Locking nur bei einer Änderung die Tabelle gesperrt wird, minimiert sich so die Anzahl der Sperren. Der pCP bleibt im Speziellen besser benutzbar. Dazu werden je nach Aktion verschiedene Granularitäten für die Sperre gewählt, d.h. es können entweder ganze Tabellen gesperrt werden, (z.B. Verschiebeoperation) oder nur einzelne Datensätze innerhalb der Tabelle. In den meisten Fällen werden diese nur auf einzelne Elemente gesetzt. Nur bei Verschiebeoperationen werden größere Bereiche gesperrt, da diese mehrere Einträge und Checklisten betreffen können.

5.1.1 Konfliktlösung

Zur Konfliktlösung können generell zwei Verfahren definiert werden. In jedem Fall wird zunächst auf dem pCP eine automatische Zusammenführung der Daten versucht (siehe Kapitel 5.1.1). Kommt es hier zu nicht lösbaren Konflikten, sollen auf dem pCC Möglichkeiten zur manuellen Zusammenführung (siehe Kapitel 5.1.2) zur Verfügung gestellt werden.

Automatische Konfliktlösung

Wie in Abbildung 5.3 dargestellt, werden zur automatischen Konfliktlösung und Vermeidung ein Changelog, die Versionierung und Sperrfunktionen auf dem pCP verwendet. In Abbildung 5.4 wird der Algorithmus zur Konfliktlösung auf Serverseite dargestellt.

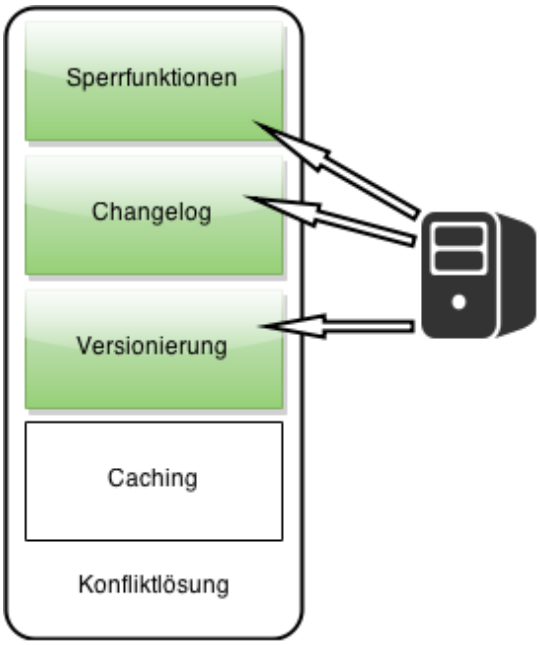


Abbildung 5.3: Konfliktlösung: pCP (genauer siehe Seite 54)

Grundlegender Ablauf: Zunächst wird geprüft, ob aktuell Sperren auf den Datensatz (z.B. Checklisten Eintrag) vorhanden sind. Ist dies nicht der Fall, wird die übertragene *Versions-ID* überprüft. Entspricht diese *Versions-ID* nicht der aktuellen, wird im Changelog nach dieser gesucht. Wird sie gefunden, werden alle Änderungen zwischen der aktuellen und der alten *Versions-ID* zurückgegeben. Der pCP kann nun im nächsten Schritt prüfen, ob die vom pCC übertragenen Änderungen in Konflikt mit einer der schon vorhandenen Änderungen stehen. Dazu überprüft er in

allen vorhanden Änderungen, ob dasselbe Feld wie im aktuellen Fall bearbeitet wurde. Ist dem so, wird die Anfrage abgelehnt und die neueste Ressource zurückgeschickt. Der pCC muss sich nun um die manuelle Konfliktlösung kümmern. Konnten jedoch keine Konflikte gefunden werden, wird eine Sperre auf das Element angefordert und die Änderung durchgeführt.

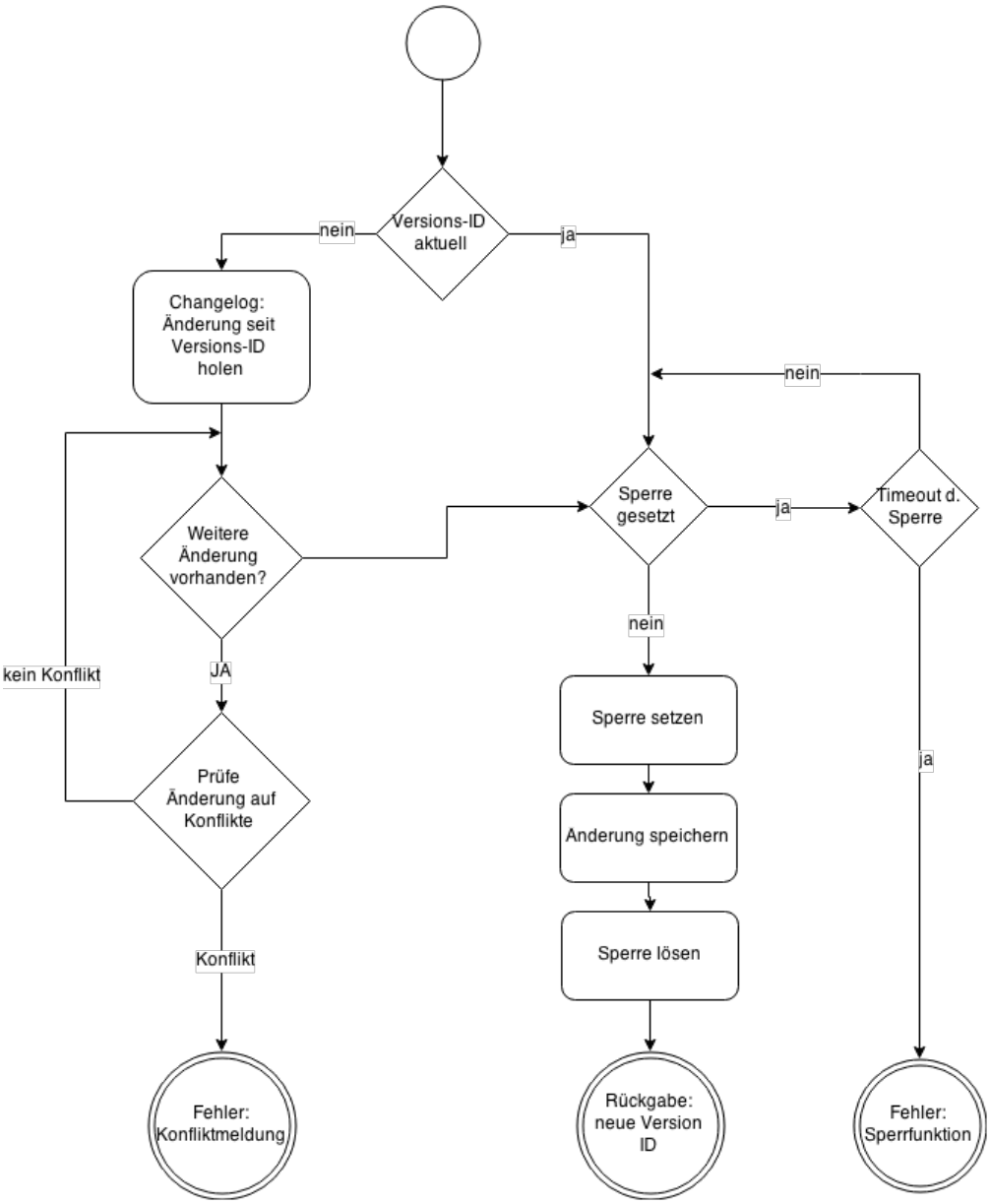


Abbildung 5.4: Konfliktlösung Serverseite 1

5.1.2 Durchführung

Um die Abläufe der Konfliktlösung zu verdeutlichen, werden einige Beispiele aufgezeigt. Die in Abbildung 5.5 gekürzt dargestellte Checkliste ist dabei jeweils der Ausgangspunkt. Die Einträge der Checkliste werden nur mit ID, Name und Versions-ID dargestellt. Dieser Zustand ist auf pCP und pCC vorhanden.

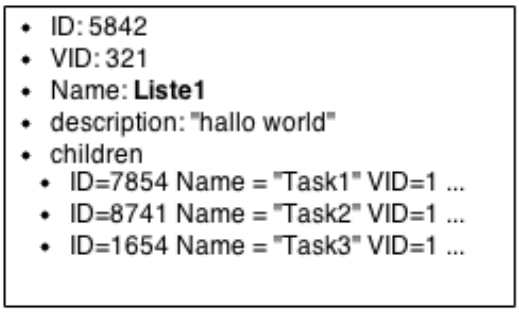


Abbildung 5.5: Datenstruktur: Checkliste

Beispielablauf 1

In Abbildung 5.6 ist der beispielhafte Ablauf einer automatischen Konfliktlösung aus Sicht des pCCs dargestellt. Dabei wurden verschiedene Felder bearbeitet und die Änderungen können zusammengeführt werden. Eigentlich müsste Bob nach der Änderung von Alice erst synchronisieren, um die neuesten Änderungen zu erhalten, da für jeden Schreibzugriff die Versionsnummer inkrementiert wird. Durch die automatische Konfliktlösung entfällt dieser Schritt in diesem Fall. Bei erfolgreicher Änderung durch den pCC wird immer die neue *Versions-ID* zurückgegeben. Der pCC muss diese bei sich eintragen, da es ansonsten im nächsten Schritt zu Inkonsistenzen kommt. Auf der rechten Seite wird jeweils die Liste dargestellt, die Alice und Bob zurückgegeben wird. Dabei sieht man, dass Bob eine Versionsnummer überspringt und die Änderungen von Alice direkt erhält.

Auf Seite des pCP ist dieser Fall durch die Konfliktlösung einfach aufzulösen. Die verschiedenen Abläufe für Alice und Bob werden in Abbildung 5.7 dargestellt.

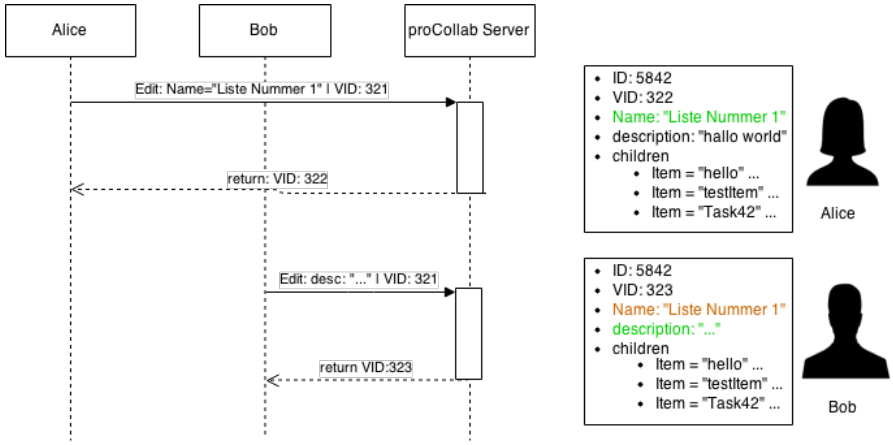


Abbildung 5.6: Konfliktlösung 1: pCC

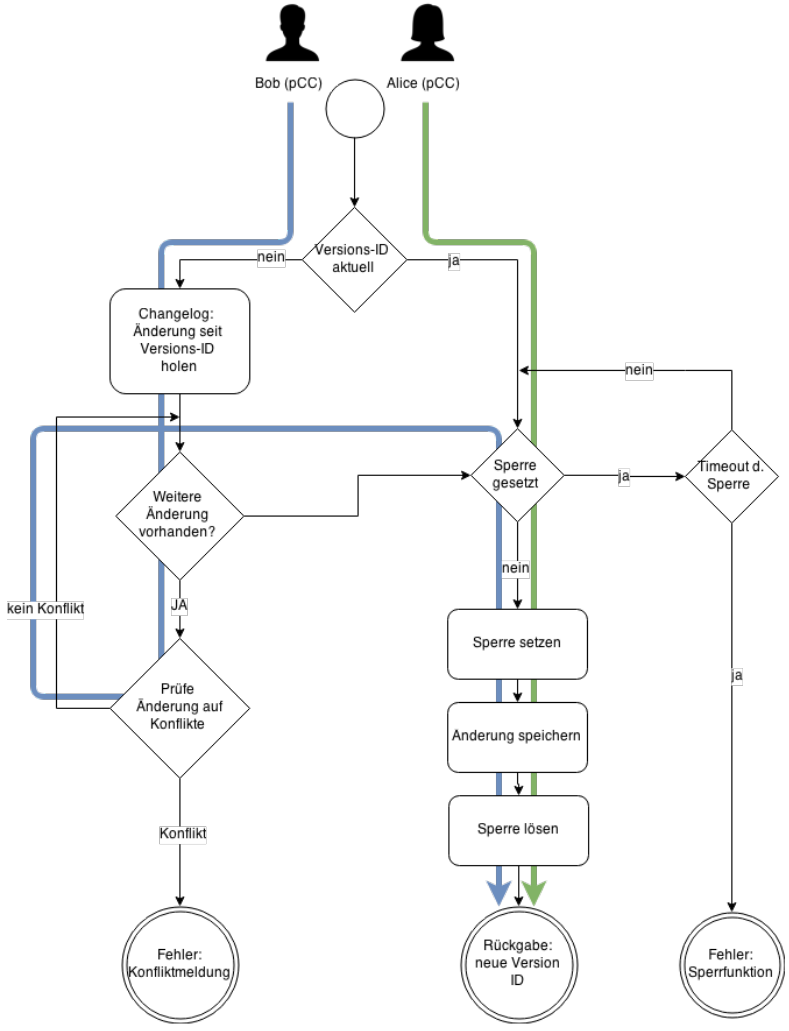


Abbildung 5.7: Konfliktlösung 1: pCP

Beispielablauf 2

Im diesem Beispiel kommt es zu einem nicht automatisch lösbaren Konflikt auf Basis von AF2 (siehe Kapitel 4.2). Alice und Bob bearbeiten nacheinander das gleiche Feld in der Checkliste. Da Bob die parallelen Änderungen von Alice nicht erhalten hat, werden seine zunächst ablehnt. Bob bekommt die neueste Version der Liste zurückgegeben und hat nun die Möglichkeit zur manuellen Konfliktlösung (siehe Kapitel 5.1.2). Seine Änderungen gehen dabei nicht verloren. Auf der rechten Seite werden die entsprechenden Rückgaben an die Clients angegeben.

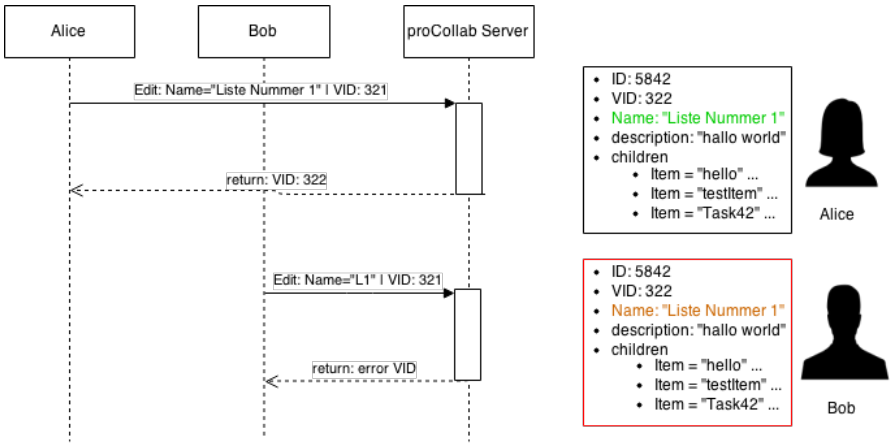


Abbildung 5.8: Konfliktlösung 2: pCC

Auf Seite des pCP unterscheidet sich der Ablauf nun für Bobs Anfrage (Abbildung 5.9). Bob stellt eine Anfrage mit veralteter Versionsnummer da er die Änderung von Alice noch nicht erhalten hat. Beim Durchgehen der Änderungen kommt es zu einem Konflikt mit der vorangegangenen Änderung von Alice. Die automatische Konfliktlösung kann hier nicht angewendet werden. Bob wird aus diesem Grund ein Fehler zurückgegeben und damit die manuelle Konfliktlösung auf dem pCC angestoßen.

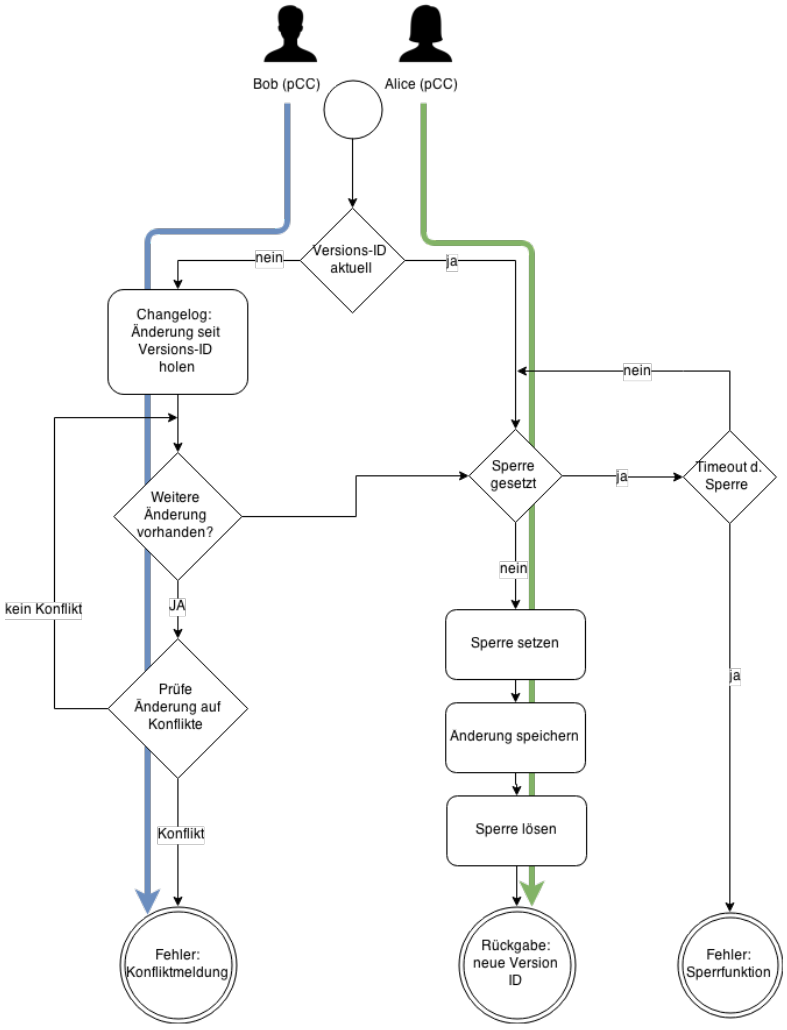


Abbildung 5.9: Konfliktlösung 2: pCP

Beispielablauf 3

In diesem Beispiel bearbeiten Alice und Bob zeitgleich verschiedene Teile der Checkliste. Es sollen die gleichen Änderungen wie in Beispiel 1 durchgeführt werden. Für den Ablauf auf dem pCP gilt weiterhin Abbildung 5.6,aber es kann für Bob zu Verzögerungen kommen. Alice kann als Erste die Sperre auf das Feld setzen. Da Bob nun eine Änderung durchführen will, während das Feld noch gesperrt ist, kommt es zu einem geänderten Ablauf auf dem pCP. Wird die Sperre innerhalb eines definierten Zeitraums wieder gelöst

und Bob kann seine Änderung durchführen, wird die Anfrage korrekt ausgeführt. Ist die Sperre jedoch zu lange gesetzt, bricht die Ausführung ab. Dieser wird in Abbildung 5.10 dargestellt.

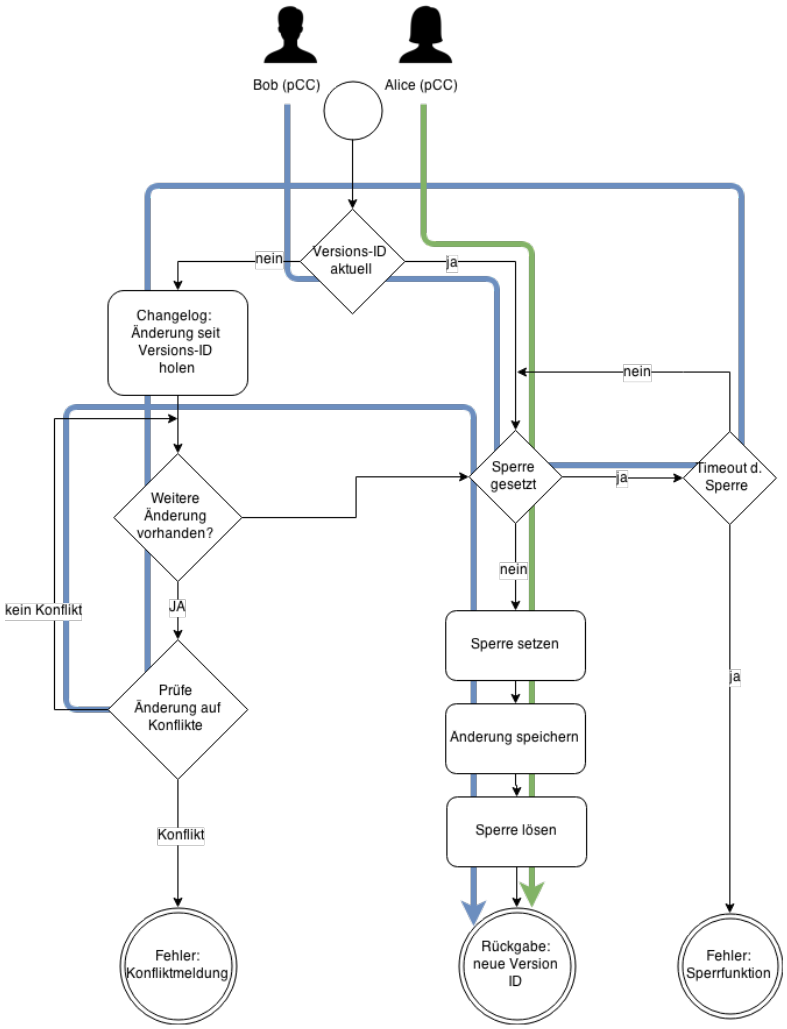


Abbildung 5.10: Konfliktlösung 2: Server

Manuelle Konfliktlösung durch den Client

Nicht immer ist es für den pCP möglich, Konflikte automatisch zu lösen (siehe Kapitel 5.1.1). Ist dies der Fall, sollen Möglichkeiten zur manuellen Konfliktlösung durch den Benutzer auf den Client gegeben werden. Dafür sind Änderungen und Erweiterungen an der Benutzeroberfläche des pCC notwendig. Das in Abbildung 5.11 dargestellte Mockup wird dabei als Ausgangspunkt verwendet. Es ist an die Tabletversion [Köl13] des vorhandenen pCC angelehnt und zeigt die Bearbeitungsansicht eines Checklisten Eintrags.

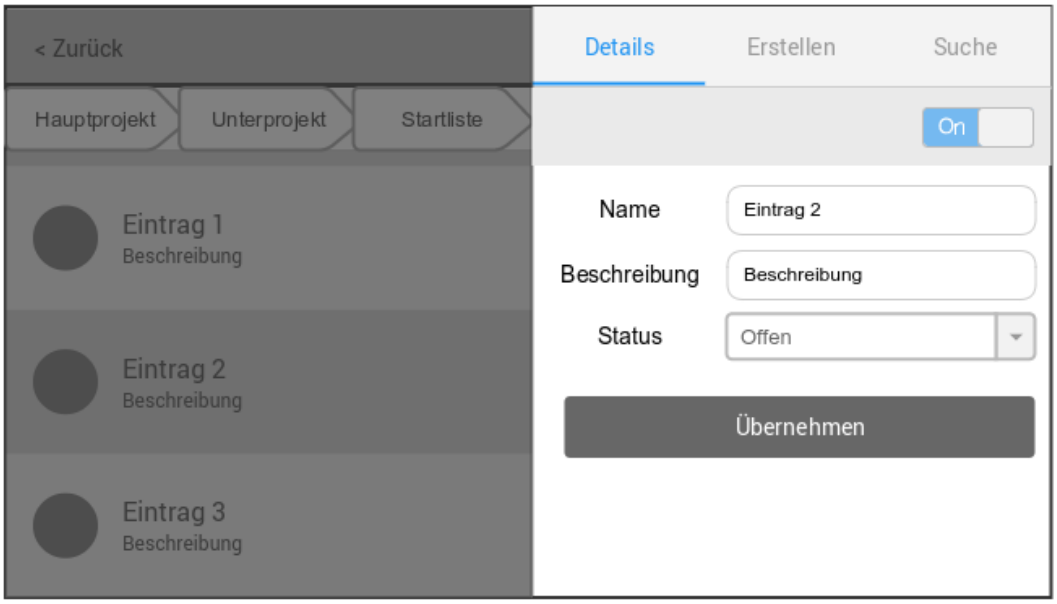


Abbildung 5.11: Mockup der Bearbeitungsansicht eines Eintrags

Wie in Abbildung 5.11 illustriert werden innerhalb der Detailansicht zwei Icons eingefügt, die den Benutzer über den aktuellen Status des Elements informieren. Ist das Element nur lokal gespeichert, wird das rechte Symbol nicht angezeigt. Ist es dagegen in gleicher Version auch auf dem Server vorhanden, soll das rechte Symbol angezeigt werden (Abbildung 5.12).

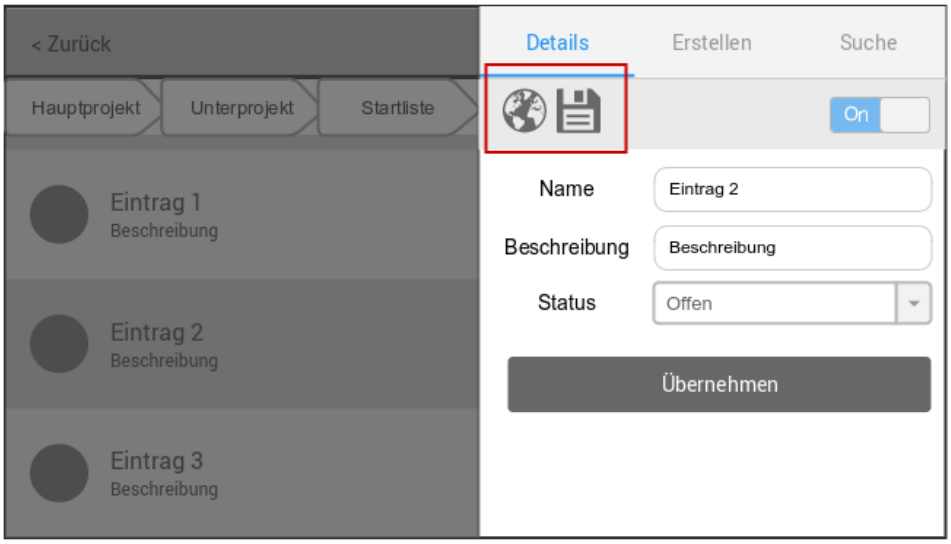


Abbildung 5.12: Neue Icons der Bearbeitungsansicht

Wird dem pCC nun vom pCP ein Konflikt mitgeteilt, ändert sich die Detailansicht. Die Attribute mit Konflikten werden markiert. Der Benutzer hat nun die Möglichkeit diese anzuklicken und die Änderungen, die erhalten bleiben sollen, auszuwählen. Die manuelle Konfliktlösung sollte dabei nur bei bestehender Internetverbindung möglich sein, da es sonst leicht zu weiteren Konflikten kommen kann. Über die Status-Icons wird der Benutzer über den Status der Datenübertragung informiert. In Abbildung 5.13 ist der komplette Ablauf einer manuellen Konfliktlösung dargestellt. In diesem Beispiel wurde dabei ein Konflikt bei dem Namensfeld eines Eintrags festgestellt und gelöst. Der Benutzer hat in a) zunächst den Parameter mit dem Konflikt ausgewählt. Daraufhin öffnet sich der in b) angezeigte Auswahldialog, um die gewünschte Änderung auszuwählen. Der Benutzer hat die Änderung der Serverseite ausgewählt und seine eigene verworfen c).

Auch auf dem Startbildschirm der Applikation werden einige Anpassungen vorgenommen. In der oberen Leiste wird neben dem Menüpunkt **Benutzerkonto** ein Statusicon eingefügt, um den aktuellen Synchronisationsstatus anzuzeigen (siehe a in Abbildung

5.14). Es wird zwischen drei Stufen unterschieden. Außerdem kann von hier aus der genaue Staus abgefragt werden (siehe b in Abbildung 5.14).

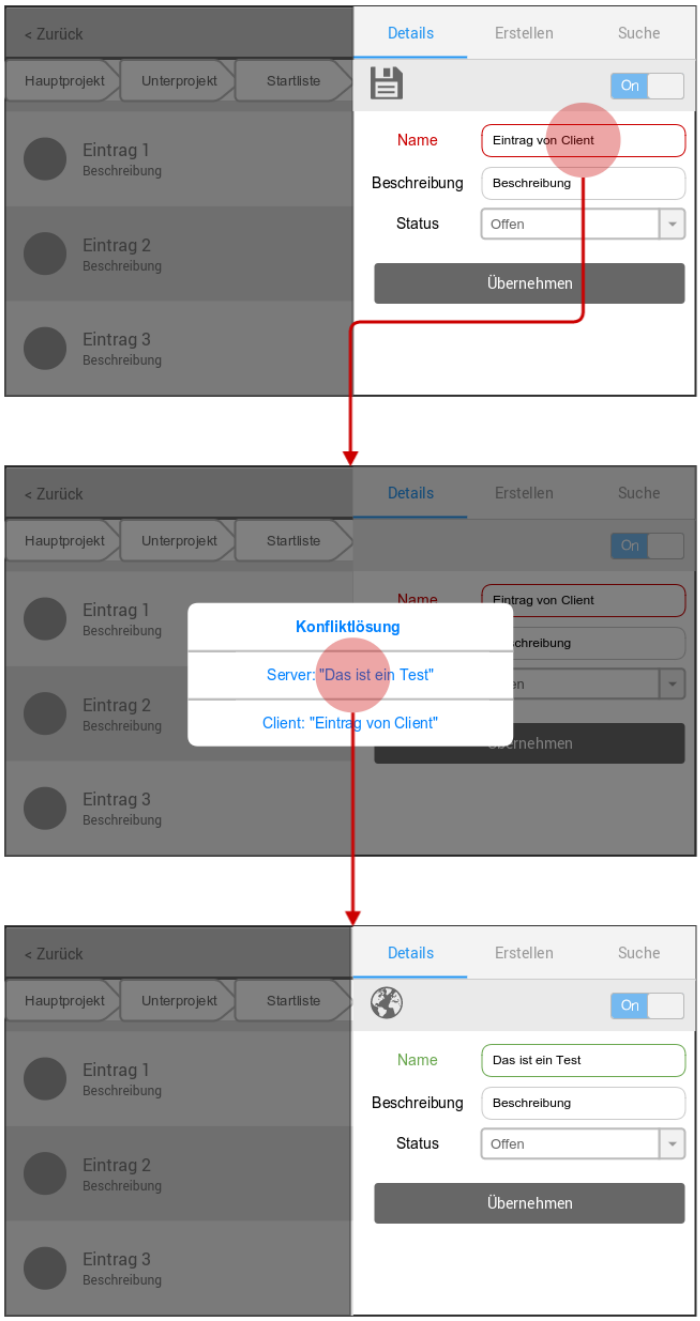


Abbildung 5.13: Mockup: Ablauf der manuellen Konfliktlösung

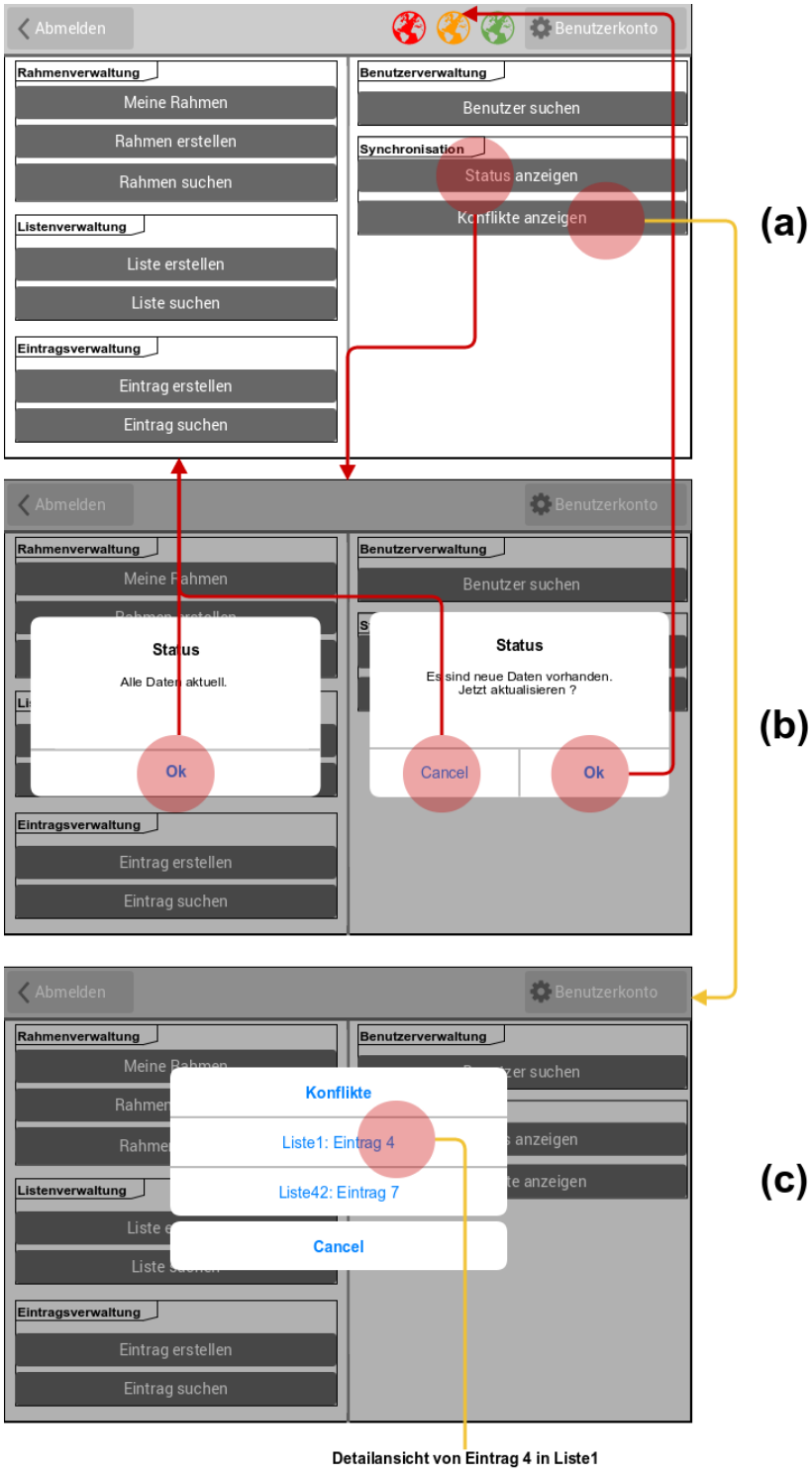


Abbildung 5.14: Mockup: Anpassungen des Startbildschirms

Wie in Abbildung 5.14 a) dargestellt, wird außerdem, ein neuer Menüpunkt **Synchronisation** hinzugefügt. Durch diesen ist es möglich, den aktuellen Synchronisationsstatus abzurufen und zu aktualisieren. Ebenso kann eine Übersicht der vorhandenen Konflikte dadurch aufgerufen werden.

6

Implementierung

Für die Implementierung des in Kapitel 5 vorgestellten Konzepts zur Synchronisation und Fehlertoleranz werden zunächst die verwendeten Technologien besprochen. Auf Basis dieser werden für den pCP und den pCC anschließend Auszüge aus den Implementierungen diskutiert.

6.1 Datenmodell

Die Datenmodelle des pCC und des pCP müssen erweitert werden, um ein Attribut für die Versionierung hinzuzufügen. Da aufgrund von verschiedenen Programmierparadigmen nicht exakt die gleichen Datenstrukturen auf Client und Server verwendet werden, müssen beide separat bearbeitet werden.

proCollab-Prototyp In Abbildung 6.1 ist das aktuelle Datenmodell des pCP vereinfacht dargestellt. Die markierte Ressource 'BaseEntity' ist das Basiselement und muss um das Feld *Versions-ID* erweitert werden, dieses ist fett markiert. Durch die Vererbung erhalten alle unteren Elemente ebenfalls das Feld *Versions-ID* .

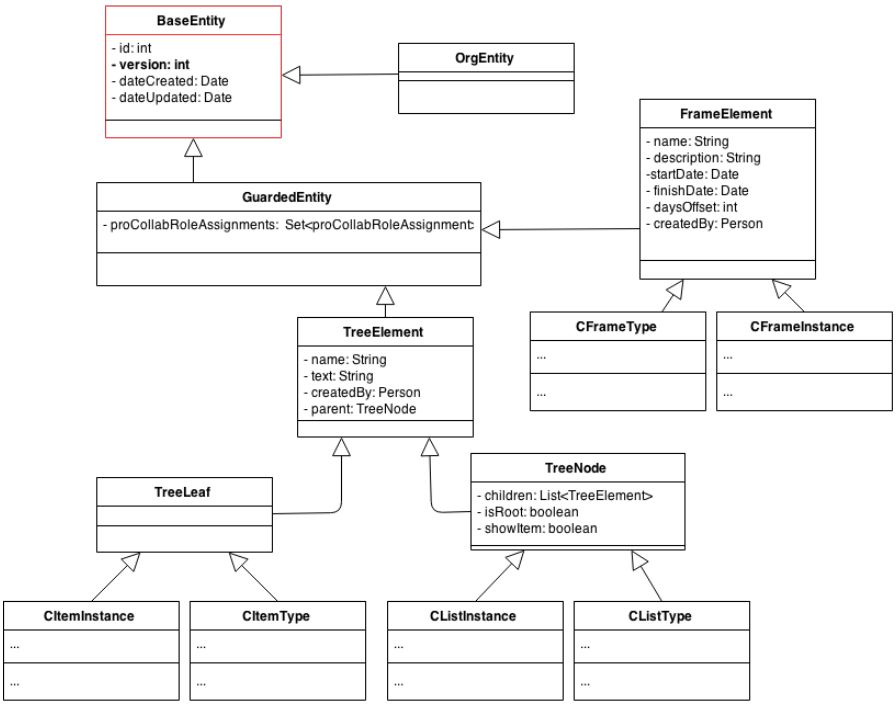


Abbildung 6.1: *proCollab -Server* : Aktuelles Datenmodell

proCollab-Client Auf dem pCC müssen mehrere Ressourcen angepasst werden, als auf dem Server. Die in der Abbildung 6.2 markierten Objekte müssen um das Versions-ID-Feld erweitert werden. Durch die Vererbung besitzen nun alle Elemente das Feld *Versions-ID* .

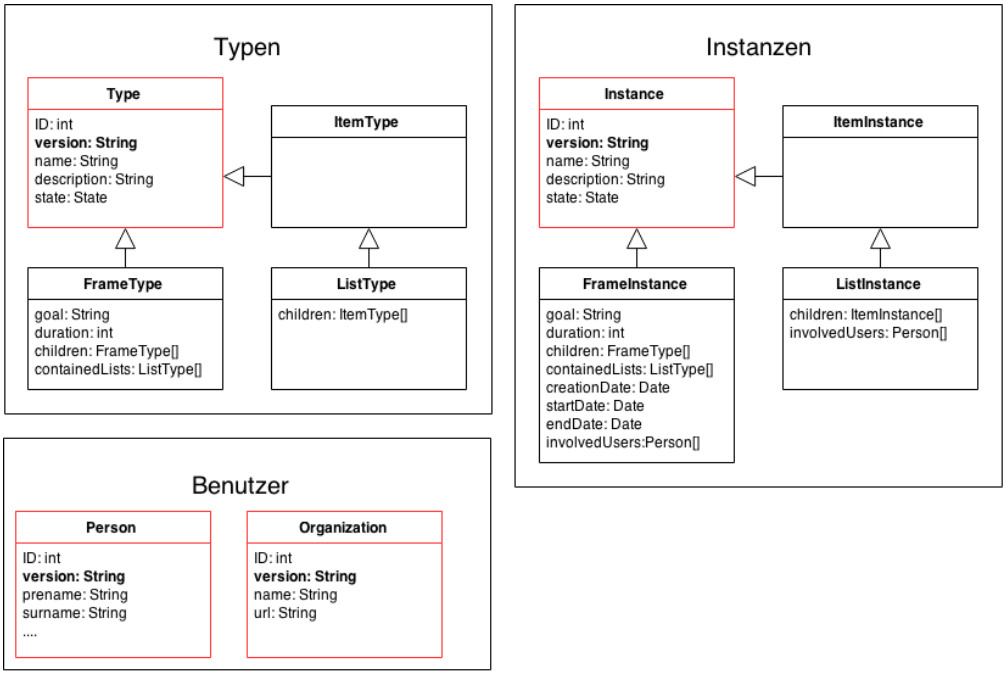


Abbildung 6.2: *proCollab -Client* : Aktuelles Datenmodell

Durch die Anpassungen der Datenstrukturen des pCP und des pCC ist sichergestellt, dass jede Ressource in diesem System, neben ihrer eindeutigen ID, auch noch ein Versions-Feld besitzt. Dem pCP ist es nun möglich, verschiedene Versionen der gleichen Ressource zu vergleichen und in Relation zu setzen.

6.2 Verwendete Technologien

In diesem Kapitel werden die verwendeten Technologien vorgestellt, die dazu benutzt werden, das in Kapitel 5 vorgestellte Konzept zu realisieren.

6.2.1 JavaEE

Es werden verschiedene in der JavaEE Spezifikation enthaltene Komponenten verwendet. Im Speziellen sind das *JAX-RS*, *JPA*, *Annotationen* und *Interceptoren* [Jen06]. Die verwendeten Komponenten sind in Abbildung 6.3 blau markiert, darüber hinaus sind weitere von pCP genutzte, aber für diese Arbeit nicht relevanten, Komponenten grau markiert. Als Anwendungsserver wird JBoss WildFly verwendet [wil].

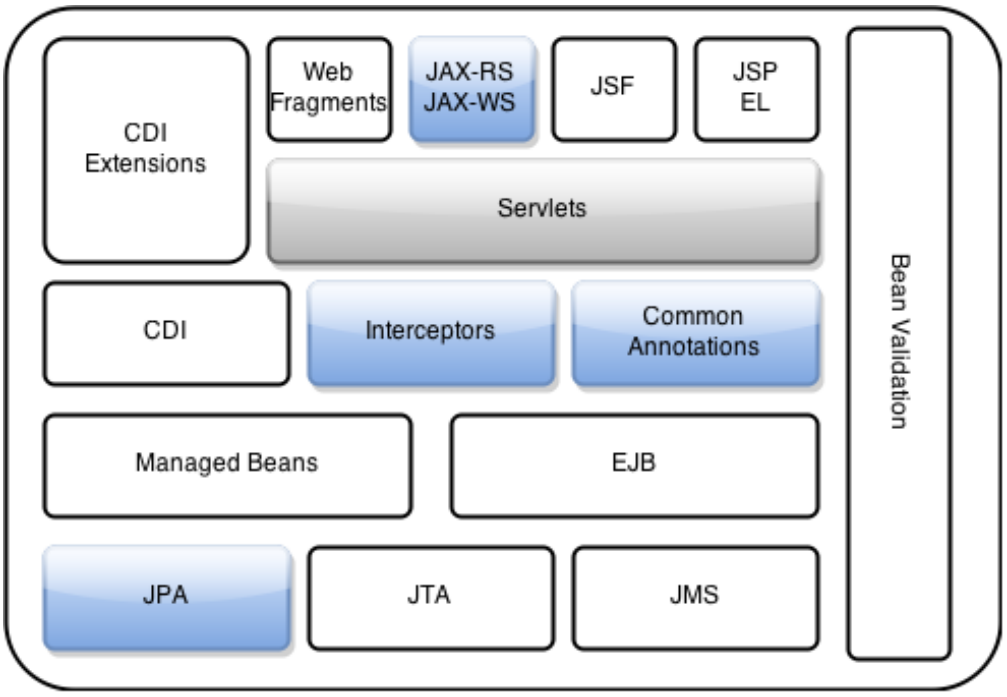


Abbildung 6.3: Java EE Komponenten nach [jav]

Annotation

Annotationen sind Sprachelemente in Java, die es erlauben zu dem eigentlichen Quellcode weitere Metaebenen hinzuzufügen [jav]. Es können dabei zum einen von Java mitgelieferte Annotationen wie *Deprecated* oder *Override* verwendet werden. Viele Komponenten innerhalb von JavaEE, wie JAX-RS, werden darüber hinaus mittels Annotation gesteuert [Jen06]. Ein Beispiel für die Verwendung von Annotationen wird in Listing

6.3 angegeben. Im Falle des pCP kommen konkret die Annotation *@AroundInvoke* und *@Interceptors* zum Einsatz um die bestehende Funktionalität, mithilfe von *Interceports* (siehe Kapitel 6.2.1) zu erweitern.

Interceptor

Durch den Einsatz des Interceptoren-Konzepts ist eine einfache Erweiterung vorhandener Komponenten möglich. Dabei werden Klassen als spezielle Module definiert, die in den Ablauf anderer eingehängt werden können. In Abbildung 6.4 ist eine solche Erweiterung schematisch dargestellt. Der normale Aufruf geht von Komponente A zu Komponente B. Durch den Einsatz des Interceptoren-Konzepts kann dieser Aufruf nun über die Komponente C umgeleitet werden. Darauf können weitere Funktionalitäten oder Prüfungen der Aufrufparameter durchgeführt werden. Nachdem der Interceptor ohne Fehler durchlaufen wurde, wird der Aufruf weiter an das eigentliche Ziel, also Komponente B weitergegeben. Bei Fehlern innerhalb der Interceptor Komponente wird die Anfrage abgebrochen. Um Interceptoren anzuwenden werden die oben vorgestellten Annotationen benutzt.

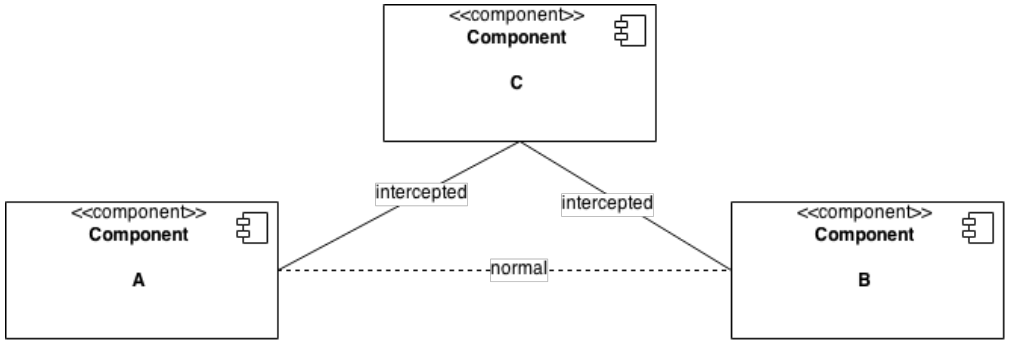


Abbildung 6.4: Interceptor Schema

6.2.2 Java Persistence API - Hibernate

Zur Speicherung der Daten im pCP wird die Java Persistence API (JPA) und Hibernate verwendet. Dabei werden die Java Objekte durch die objektrelationale Abbildung

(ORM) in einer Datenbank gespeichert. Unter ORM versteht man dabei die automatisierte Abbildung von Objekten auf ein relationales Datenbank-Management-System [BK06]. Hibernate ist ein Open-Source Framework, das JPA verwendet um Daten persistent zu speichern. Im Speziellen bietet es Funktionen, wie Transaktionsverfahren oder Suchfunktionen, die durch Annotationen einfach genutzt werden können [BK06].

6.2.3 REST-Schnittstelle und JAX-RS

Representational State Transfer, kurz REST, genannt ist ein Architekturkonzept für den Datenaustausch in verteilten Systemen. Es wird vor allem zur Maschine-Maschine-Kommunikation einsetzt und basiert auf den grundlegenden Mechanismen des HTTP-Protokolls [Bur09]. Dies hat den Vorteil, dass große Teile der benötigten Infrastruktur, wie Anwendungsserver und kompatible Clients, bereits vorhanden sind. Nach [Til09] ergeben sich bei REST fünf Kernprinzipien:

Eindeutige Identifikation Eindeutige Identifikation besagt, dass alle Ressourcen einen eindeutigen Namen oder ID besitzen müssen. Bei der REST-Schnittstelle wird dabei die URI verwendet [Bur09].

Verknüpfungen Verknüpfungen sind im Kontext von REST einfach HTTP-Links. Innerhalb einer Ressource können ein oder mehrere Links auf andere Ressourcen gesetzt sein. Über diesen Link können weitere Informationen abgerufen werden [Til09].

Standardmethoden Da die REST-Schnittstelle auf dem HTTP-Protokoll aufbaut, werden die Standard Methoden von HTTP zum Nachrichtenaustausch verwendet. Neben den bekannten Methoden *GET* und *POST* werden auch *HEAD*, *OPTIONS*, *DELETE* und *PUT* verwendet [Bur09]. Da bei allen Ressourcen das gleiche Interface benutzt gilt, wird zum Abfragen immer *GET* verwendet. Mit *POST* werden neue Einträge angelegt und mit *PUT* aktualisiert. Mit *DELETE* werden Einträge gelöscht. Mit *OPTIONS* und *HEAD* können zum einen unterstützte Methoden der aktuellen Ressource, zum anderen Metadaten, abgefragt werden.

Unterschiedliche Repräsentation Daten, die über eine REST-Schnittstelle abgefragt werden, können unterschiedlich repräsentiert werden.

Statuslose Kommunikation Die Kommunikation der REST-Schnittstelle erfolgt komplett statuslos. Das heißt, es werden auf dem Server keine Zustandsinformationen zu einzelnen Clients gehalten. Die Zustandsverwaltung muss entweder über den Zustandswechsel von Ressourcen realisiert werden oder komplett auf Clientseite erledigt werden.

JAX-RS

Die Java API for RESTful Web Services Spezifikation ist ebenfalls Teil von JavaEE und ermöglicht die Entwicklung von RESTful Webservices mittels Annotationen. Das Open-Source-Projekt Jersey [jer] ist die Referenzimplementierung der Spezifikation. Daneben existieren einige weitere Implementierung wie Restlet [res] oder Apache Wink [win].

6.2.4 PhoneGap

Das Open-Source Framework PhoneGap [pho] wird zur Entwicklung des proCollab-Clients verwendet. Mit PhoneGap können plattformunabhängige mobile Applikation erstellt werden. Dabei werden die Webtechnologien HTML5, CSS und JavaScript zur Entwicklung eingesetzt. Durch abstrahierte Schnittstellen kann auf verschiedene Hardwarekomponenten wie Datenspeicher oder Kamera zugegriffen werden [AGL10].

6.2.5 JQuery Mobile

Zur dynamischen Gestaltung der View Komponente wird JQuery Mobile eingesetzt. Dieses ist ebenfalls als Open-Source verfügbar und liefert neben vielen nützlichen Operationen um die Struktur der HTML-Seite zu verändern, eine große Anzahl von Dialog-

und Bedienelementen mit, die auf verschiedene Gerätetypen und Bildschirmgrößen angepasst sind [jqu].

6.3 Auszüge aus der Implementierung

Im Folgenden werden Auszüge aus den Implementierungen zur Umsetzung der Konzeption für den pCP und den pCC angegeben. Dabei werden verschiedene relevante Aspekte im Detail dargestellt.

6.3.1 Erweiterungen des proCollab-Prototypen

Erweiterung der REST-Schnittstelle

Der pCP muss eine REST-Schnittstelle bereitstellen über die Änderungen, die nach einem bestimmtem Zeitpunkt getätigt wurden, abgefragt werden können. Durch diese Funktion kann der pCC seinen lokalen Cache aktualisieren (siehe Kapitel 5.1). Dabei wird eine Liste von URLs zu den aktualisierten Elementen zurückgegeben.

Zunächst muss das Interface *SynchronisationInterface* für die neue REST-Schnittstelle angelegt werden. Dieses definiert lediglich die Funktion *getChanges*. In Listing 6.1 ist das Interface und in Listing 6.2 die Implementierung angegeben.

Listing 6.1: Schnittstelle Synchronisation

```
1 public interface SynchronizationInterface {
2   public Response getChanges(String sessionID, int timestamp);
3 }
```

Listing 6.2: RestSynchronizationManager

```
@Path("/synchronization")
@Consumes({ "application/json" })
@Produces({ "application/json" })
public class RestSynchronizationManager implements
    SynchronizationInterface {

private static FrameInstanceManager frameManager =
    FrameInstanceManager.getFrameInstanceManager();
  ...
  @Override
  @GET
  @Path("/{timestamp}")
  public Response getChanges(@CookieParam("sessionID") String
      sessionID, @PathParam("timestamp") int timestamp);
    Person person = authenticationManager.getLoggedPerson(sessionID);
    ...
    ArrayList<CFrameInstance> framesFromUser =
        frameManager.getFrameInstancesFromUser(person.ID);
    ArrayList<String> urls = new ArrayList<String>();

    // Alle Frames und Unterelemente durchgehen
    forearch(CFrameInstance i : framesFromUser){
      if(i.dateUpdated > timestamp){
        urls.add("frame/"+i.id);
      }

      // Checklisten und Items rekursiv durchgehen
      ...
    }
    return Response.status(Status.OK).entity(urls).build();
  }
}
```

Mit den Annotationen in den Zeilen eins bis drei wird JAX-RS konfiguriert. So wird mit der *@PATH*-Annotation der Pfad, unter dem die Ressource erreichbar sein soll, angegeben. Außerdem wird das gewünschte Datenformat auf JSON gesetzt.

In den Zeilen neun bis elf wird anschließend die Methode *getChanges* mit der HTTP-Funktion GET verknüpft. Wobei noch der Parameter *timestamp* als Pfadvariable mitgegeben werden muss. Dieser wird der Funktion übergeben.

Im Rumpf der Methode *getChanges* werden schließlich alle Rahmen-Instanzen abgerufen, an denen der Nutzer beteiligt ist. Bei diesen wird das Änderungsdatum mit dem übertragenen Zeitstempel verglichen. Wird dabei eine Änderung erkannt, wird die URL der Ressource in die Liste der Änderungen eingetragen. Anschließend werden alle Kind-Elemente des Rahmens rekursiv durchlaufen und auf Änderungen überprüft. Am Ende wird die Liste der Änderungen an den Client übertragen.

Erweiterung der Datenstruktur und des Datenzugriffsobjekt

Auf Seite des pCP wird nun die Klasse *BaseEntity* um das Feld *version* vom Datentyp Integer erweitert (Listing 6.3). Durch die Annotation **@Version** wird Hibernate mitgeteilt, dass in diesem Feld die Versionsnummer der Ressource gespeichert werden soll. Gleichzeitig wird dadurch das *optimistic locking* aktiviert. Durch die Annotation *@Temporal(TemporalType.DATE)* der Variable *dateUpdated* wird bei jeder Änderung der Zeitstempel aktualisiert.

Listing 6.3: BaseEntity: Version-Id Feld.

```
1 public abstract class BaseEntity implements Serializable {
2   ...
3   private int id
4
5   @Version
6   private Integer version;
7
8   @Temporal(TemporalType.DATE)
9   private Date dateUpdated;
```

Als nächstes werden zwei Interceptoren (siehe Kapitel 6.2.1) implementiert, um die Funktionalität für das Erstellen des Changelogs und der Konfliktlösung bereit zu stellen. Diese werden anschließend an ein generisches Datenzugriffsobjekt (DAO) angehängt. Dieses DAO ist für die Speicherung der Daten verantwortlich. Im Speziellen muss im pCP die Klasse DaoImpl erweitert werden. Die verschiedenen Teile, die implementiert wurden, sind in Abbildung 6.5 grün markiert. Der grau markierte Bereich wird von Hibernate bereit gestellt.

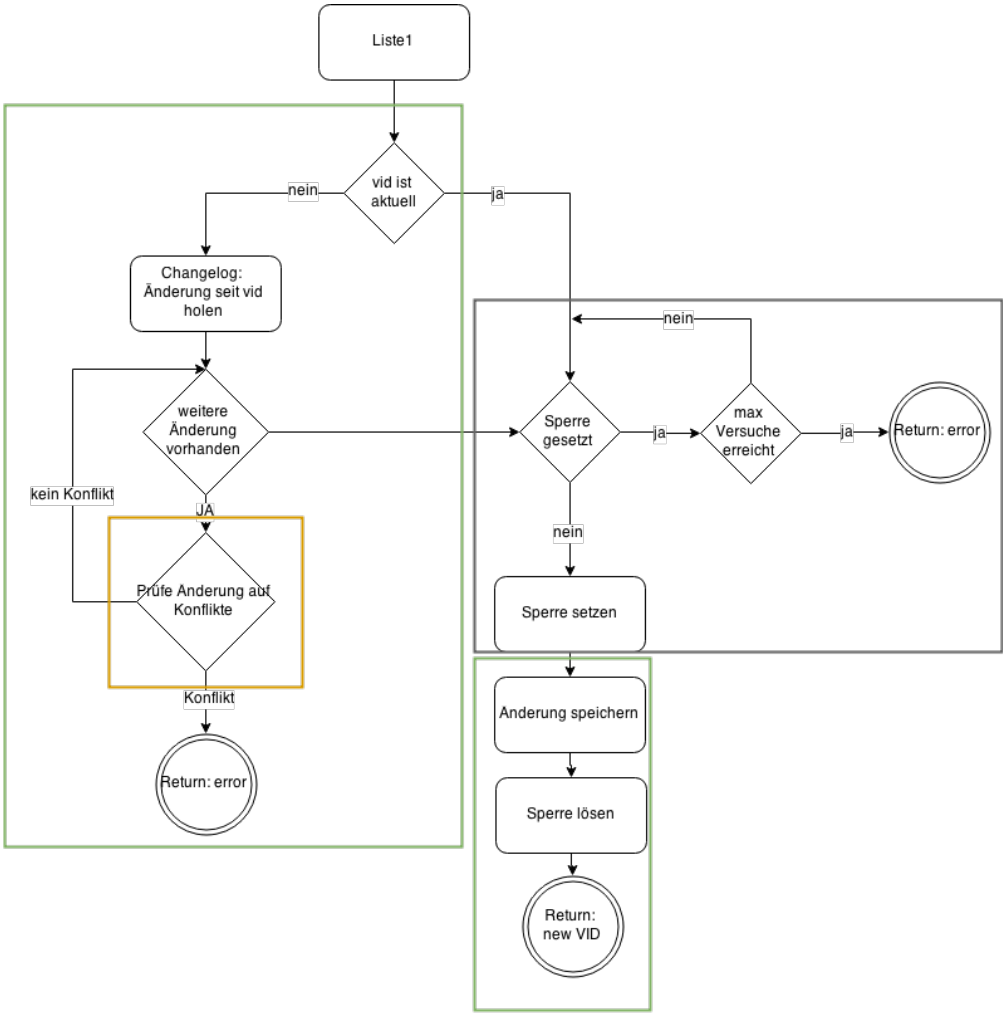


Abbildung 6.5: Konfliktlösung Implementierung

Der **MergeConflictsInterceptor** implementiert den grün markierten Algorithmus aus Abbildung 6.5. In Listing 6.4 wird dieser aus Gründen der Verständlichkeit als Pseudo-Code angegeben. Die Funktionen *getChanges*, *isMergeable* und *mergeEntity* erfüllen folgende Aufgaben:

- **getChanges**: Die Methode fragt alle Änderungen ab, die zwischen der übertragenen und der aktuellen Versions-ID getätigt wurden. In Index 0 der Rückgabe befindet sich dabei das Ausgangselement.
- **isMergeable**: Die Methode prüft, ob die übergebenen Datensätze zusammengeführt werden können. Der dritte Parameter ist dabei das Ausgangselement. Die Funktion implementiert den orange markierten Teil in Abbildung 6.5
- **mergeEntity**: Die Methode führt die Änderungen aus dem zweiten Datensatz mit dem ersten zusammen. Dazu wird der Datensatz feldweise durchschritten und auf Änderungen überprüft.

Mit der Annotation in Zeile 2 wird die Methode *mergeConflicts* als Interceptor definiert. Durch den Parameter *InvocationContext ctx* bekommt die Methode den Kontext der Ursprungsmethode übergeben.

Innerhalb der Methode werden zunächst die an die Ursprungsmethode übergebenen Parameter zur weiteren Verarbeitung zwischengespeichert. Da der MergeConflictsInterceptor an den DAO angehängt werden soll, ist dies nur die zu verarbeitende Ressource. Im nächsten Schritt wird abgefragt, ob Änderungen für die übertragene Ressource vorhanden sind. Dazu wird überprüft, ob die Ressource mit einer neueren *Versions-ID* vorhanden ist.

Sind Änderungen vorhanden, wird versucht diese mit der übertragenen Ressource zusammenzuführen. Ist dies erfolgreich, wird die zusammengeführte Ressource an die Ursprungsmethode übergeben. Kommt es zu nicht lösbaren Konflikten, wird ein Fehler zurückgegeben. Dabei wird die übergebene Ressource und die Änderung die den Konflikt ausgelöst hat mitgegeben.

Listing 6.4: MergeConflictsInterceptor

```
1  public class MergeConflictsInterceptor {
2    @AroundInvoke
3    public Object mergeConflicts(InvocationContext ctx) throws
         Exception {
4      Object[] parameters = ctx.getParameters();
5      Entity entity = (Entity) parameters[0];
6
7      Entity changes[] = getChanges(entity);
8
9      foreach(Entity e : changes){
10       if(isMergeable(entity, e, changes[0]){
11         entity = mergeEntity(entity, e)
12       }else{
13         throw new NotMergeableException(parameters[0], e)
14         return null;
15       }
16     }
17     parameters[0] = entity;
18   }
19   return ctx.proceed();
20 }
```

Der zweite Interceptor, der implementiert wird, ist der **ChangelogInterceptor**. Dieser wird abgearbeitet, wenn Änderungen erfolgreich gespeichert werden konnten. Ein illustrierter Pseudo-Code ist in Listing 6.5 angegeben. Dabei ist wichtig, dass dieser Interceptor erst nach dem eigentlichen Methodenaufruf im DaoImpl ausgeführt wird. Dies wird durch Zeile 4 mit dem Kommando *ctx.proceed()* sichergestellt.

Listing 6.5: ChangelogInterceptor

```
1 public class ChangelogInterceptor {
2   @AroundInvoke
3   public Object (InvocationContext ctx) throws Exception {
4     ctx.proceed();
5     Object[] parameters = ctx.getParameters();
6     Entity entity = (Entity) parameters[0];
7     ...
8   }
```

Um die beiden Interceptoren nun zu benutzen, müssen diese noch mit den entsprechenden Methoden in der Klasse DaoImpl verknüpft werden. Dazu wird die Annotation **@Interceptors** gefolgt von den Namen der gewünschten Interceptoren an den Methoden eingefügt. Dadurch werden bei jedem Aufruf der Methoden *persist*, *update* oder *remove* auch die Methoden *mergeConflicts* und *logChanges* aufgerufen. In Listing 6.6 ist die entsprechende Annotation in den Zeilen 5, 11 und 15 zu beachten. Die Beziehungen der Klassen wird in Abbildung 6.6 dargestellt.

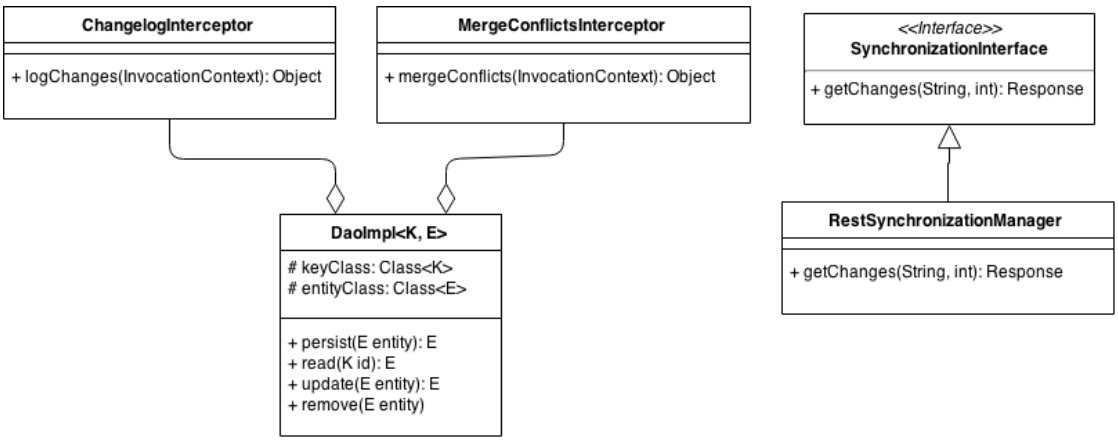


Abbildung 6.6: Beziehungen der Klassen

Listing 6.6: DaoImpl

```
1  public abstract class DaoImpl<K,E> implements Dao<K,E> {
2     ...
3    @Override
4    @Interceptors({MergeConflictsInterceptor.class,
        ChangelogInterceptor.class)
5    public E persist(E entity) ...
6    ...
7    @Override
8    @Interceptors({MergeConflictsInterceptor.class,
        ChangelogInterceptor.class)
9    public E update(E entity) ...
10
11   @Override
12   @Interceptors({MergeConflictsInterceptor.class,
        ChangelogInterceptor.class)
13   public void remove(E entity) ...
14 }
```

6.3.2 Erweiterung des proCollab-Clients

Client-Cache

Der pCC muss um einen transparenten Cache erweitert werden. Dieser ist dabei in die REST-Schnittstelle eingefügt, die zum Abfragen der Daten genutzt wird. In Abbildung 6.7 wird die Architektur mit dem Cache dargestellt.

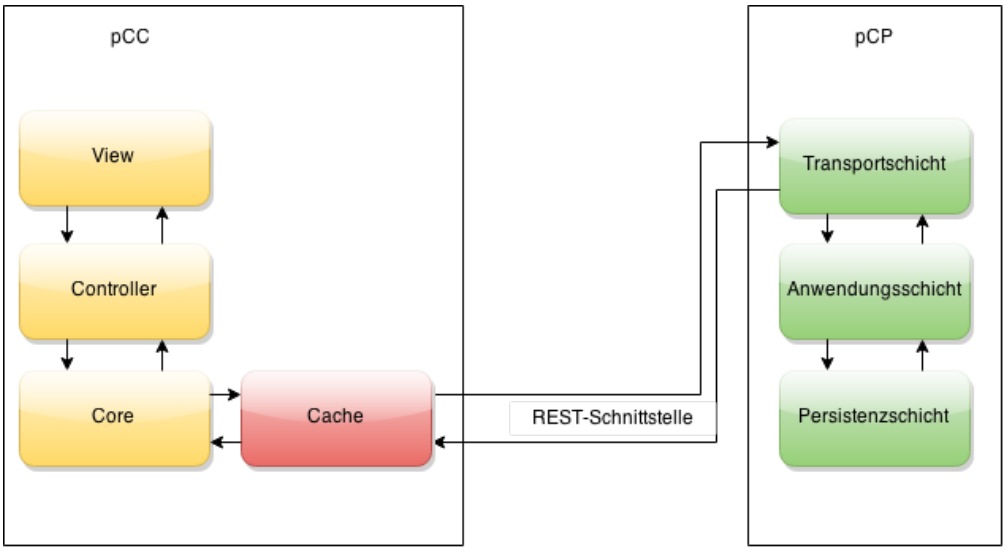


Abbildung 6.7: pCC: Cache

Der **ClientCache**, der in Listing 6.7 als Pseudo-Code angegeben ist, nutzt die *localstorage* Funktion des Frameworks PhoneGap. Wird ein neuer Datensatz zwischengespeichert, wird ihm ein Feld mit dem aktuellen Zeitstempel hinzugefügt. Dadurch ist es möglich beim Abfragen der Datensätze aus dem Cache ein maximales Alter für einen Datensatz anzugeben. Ist dieses erreicht, muss der Datensatz bei vorhandener Internetverbindung erst vom Server abgerufen werden, bevor er angezeigt werden kann.

Damit dieser Cache nun verwendet werden kann, muss er in die REST-Schnittstelle eingefügt werden. Dazu müssen die Funktionen *get*, *post*, *put* und *delete* innerhalb der Datei *rest.js*, die die REST-Schnittstelle enthält, angepasst werden. In Listing 6.8 werden die nötigen Anpassungen, beispielhaft, für die Funktion *get* gezeigt.

Listing 6.7: ClientCache

```
1  function ClientCache () {
2  this.cacheStorage = window.localStorage;
3  }
4  // ....
5  ClientCache.prototype.cacheItem = function(item) {
6    item.cachetime = new Date();
7    this.cacheStorage.setItem(item.ID, item)
8  };
9  // ....
10 ClientCache.prototype.getCachedItem = function(itemId) {
11   var now = new Date();
12   if(now - item.cachetime > maxCacheAge && connectionAvailable()){
13     return null;
14   }
15   return this.cacheStorage.getItem(itemId)
16 };
```

Der ClientCache wird als JavaScript-Prototyp implementiert. Dabei wird der von PhoneGap bereitgestellte *localstorage* verwendet, um die Daten zu speichern. In Zeile 6 wird dem Datensatz, der im Cache gespeichert wird, ein Zeitstempel hinzugefügt. Über diesen wird in Zeile 13 bis 15 überprüft, ob die Daten im Cache noch aktuell sind. Ist dies nicht der Fall oder die Daten sind im Cache nicht vorhanden, wird *null* zurückgegeben. Bei nicht vorhandener Internetverbindung, wird das Alter der Daten im Cache ignoriert. In diesem Fall sollten aber nur eingeschränkte Möglichkeiten zur Bearbeitung gegeben werden.

Wie in Listing 6.8 dargestellt, wird der Cache vor die REST-Schnittstelle geschaltet. Wird die Ressource im Cache gefunden, wird diese direkt von pCC verwendet. Ist das Ergebnis allerdings null, wird der Datensatz vom pCP abgefragt. Wurde die Anfrage erfolgreich ausgeführt, wird die zurückgegebene Ressource im Cache aktualisiert, indem die aktuelle Version im Cache überschrieben wird.

Listing 6.8: get-Funktion in rest.js

```
function get(interface, handler, handlerErr){
  var cached = cache.getCachedItem(interface);
  if(cached != null){
    handler(cached);
  }else{
    $.ajax({
      type: 'GET',
      ...
      success: function(result)
      {
        cache.CacheItem(result)
        handler(result);
      },
      error: function(response, textStatus, errorThrown)
      {
        ...
      }
  ...
```

Manuelle Konfliktlösung

Um die manuelle Konfliktlösung zu implementieren, müssen zunächst die Views, wie sie im Entwurf vorgestellt wurden, realisiert werden. Da sich durch das JQuery-Framework die DOM einfach manipulieren lässt, sind keine direkten Änderungen der bestehenden HTML-Daten der Views notwendig. Stattdessen werden die Daten beim Speichern im Controller überprüft und die vorhandene View verändert. In Listing 6.9 ist stellvertretend die Implementierung für die Speicherung eines Eintrags angegeben. Dieser wurde dabei um die Konfliktprüfung erweitert.

Listing 6.9: Controller: saveItemData

```
1  $(document).on("click", "#pageSaveItemDataBtn", function(){
2    var form = $("#pageItemInstanceDetails_Form")[0];
3    var jsonData = getFormContent(form);
4    rest.updateItemInstance(jsonData, localStorage.idii,
         function(response){
5      // Speichern erfolgreich
6      ...
7    }, function(error){
8      // Konflikte beim Speichern
9      if(error.type == "conflict"){
10       error.conflicts.forEach(function(entry) {
11           $("#pageItemInstanceDetails_Form
                 entry.name").addClass("conflict");
12           $("#pageItemInstanceDetails_Form
                 entry.name").click(manuelSolveConflict(entry))
13       });
14     }
15   );
16 });
```

Die Erweiterungen, die an dieser Methode vorgenommen wurden, befinden sich in den Zeilen sieben bis zwölf. Dabei wird überprüft, ob es beim Speichern der Ressource zum Konflikten gekommen ist. Ist dies der Fall, wird in Zeile elf das entsprechende Feld der Ressource markiert. Außerdem wird in Zeile zwölf eine Methode festgelegt, die ausgeführt wird, wenn das entsprechende Feld angeklickt wird. Dieser Methode wird als Parameter der Konflikt mitgegeben.
Die Methode erzeugt mittels JQuery einen Dialog und zeigt diesen an. Dieser Dialog verfügt, wie in Abbildung 5.13 gezeigt, über die verschiedenen Felder um zwischen der lokalen Änderung und der auf dem Server auszuwählen. Der Pseudo-Code dieser Methode ist in Listing 6.10 angegeben.

Listing 6.10: Manuelle Konfliklösung

```
function manuelSolveConflict(errorData){
 $("#conflictDialogLocalChange").html(errorData.localChange)
 $("#conflictDialogRemoteChange").html(errorData.RemoteChange)

 // open dialog
 ... function(dialogCallback){
   if(dialogCallback == local){
      $("#pageItemInstanceDetails_Form
          errorData.name").html(errorData.localChange)
   }else{
      $("#pageItemInstanceDetails_Form
          errorData.name").html(errorData.RemoteChange)
   }
 }
}
```

7

Fazit

Das letzte Kapitel der Arbeit beinhaltet eine Zusammenfassung der einzelnen Kapitel und ein abschließendes Fazit. Abschließend wird noch ein kurzer Ausblick auf mögliche zukünftige Erweiterungen gegeben.

7.1 Zusammenfassung

Am Anfang der Arbeit wurde das Forschungsprojekt proCollab ausführlich vorgestellt. Dazu wurde zunächst der Begriff Wissensarbeit erläutert und eingeführt. Im weiteren Verlauf des Kapitels lag das Augenmerk auf möglichen Einsatzgebieten für das Checklisten-Management. Diese wurden mit Anwendungsfällen veranschaulicht. Ebenso wurde die Architektur des aktuellen pCC und pCP vorgestellt und erklärt.
Anschließend wurden zunächst Grundlagen für die weitere Arbeit geschaffen. Verschie-

dene grundlegende Konzepte und Mechanismen, die für verteilte Systeme relevant sind und für unser Konzept nötig werden könnten, wurden vorgestellt.
Die Anforderungen an das Synchronisations- Konzept konnten mit Hilfe von Anwendungsfällen und deren anschließender Analyse im nächsten Kapitel herausgearbeitet werden. In den dargestellten Anwendungsfällen lag das Hauptaugenmerk dabei auf Fällen, die für die Synchronisation relevant sind.
Auf der Basis dieser Anforderungen konnte nun das eigentliche Konzept für die Synchronisation zwischen proCollab-Prototyp und proCollab-Client erarbeitet werden. Dazu wurden neben dem Algorithmus zur Konfliktlösung auch Mockups zur manuellen Konfliktlösung durch den Benutzer auf dem proCollab-Client erstellt.
Kapitel 6 befasste sich mit der Implementierung. Es wurden zunächst die verwendeten Technologien eingeführt und anschließend einige Aspekte der eigentlichen Implementierung aufgezeigt. Dabei wurden sowohl Teile des pCP als auch des pCC erläutert.

Auf Basis der vorgestellten Grundlagen, der Konzeptionierung und den diskutierten Aspekten der Implementierungen, lässt sich abschließend sagen, dass der pCP einfach um eine fehlertolerante Synchronisation der Anwendungsdaten erweitert werden kann. Durch die im Vorfeld gewählten Architektur und die verwenden Technologiegen müssen auf Client und Serverseite nur wenige Anpassungen vorgenommen werden.

7.2 Ausblick

Durch die Benutzung im Alltag kann das hier erarbeitete Synchronisations- Konzept in Zukunft evaluiert werden. Dabei sollte auf der Seite des pCC die Benutzbarkeit bei nicht vorhandener Internetverbindung überprüft werden, um zu erkennen, ob die Offlinenutzung eine wünschenswerte Funktionalität darstellt. Durch weitere Implementierungen, wie der Möglichkeit zur Definition von eigenen Regeln zur Konfliktlösung, abhängig vom aktuellen Status, Benutzer oder weiteren Faktoren, kann die Benutzbarkeit noch einmal gesteigert werden. Auf Basis der in Zukunft durch die Evaluation gewonnenen Informationen, können sich weitere Verbesserungen oder gar ganze Teilprojekte für

das proCollab-Projekt ergeben. Vor allem auf Seite der pCC gibt es noch viel Potenzial, das Nutzererlebnis zu steigern und den Wissensarbeiter noch optimaler zu unterstützen.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AGL10] ALLEN, Sarah ; GRAUPERA, Vidal ; LUNDRIGAN, Lee: *Pro smartphone cross-platform development: iPhone, blackberry, windows mobile and android development and distribution*. New York City : Apress, 2010

[ALK+02] ALTINEL, Mehmet ; LUO, Qiong ; KRISHNAMURTHY, Sailesh ; MOHAN, C. ; PIRAHESH, Hamid: Dbcache: Database Caching for Web Application Servers. In: *SIGMOD* 2002 (2002), S. 612–613

[BAH+03] BONWICK, Jeff ; AHRENS, Matt ; HENSON, Val ; MAYBEE, Mark ; SHELLENBAUM, Mark: The zettabyte file system. In: *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, 2003

[BK06] BAUER, Christian ; KING, Gavin: *Java Persistance with Hibernate*. Connecticut : Manning Publications, 2006

[Boo76] BOOTH, Grayce M.: Distributed Information Systems. In: *Proceedings of the June 7-10, 1976, national computer conference and exposition*, ACM, 1976, S. 789–794

[Bre12] BREWER, Eric: CAP twelve years later: How the rules have changed. In: *Computer* 45 (2012), Nr. 2, S. 23–29

[Bun03] BUNDESANSTALT FÜR LANDWIRTSCHAFT UND ERNÄHRUNG: Unternehmensinterne Qualitätssicherung zur Umsetzung der Verordnung (EWG) Nr. 2092/91 über den Ökologischen Landbau und die entsprechende Kennzeichnung der landwirtschaftlichen Erzeugnisse und Lebensmittel (EG-Öko-VO). (2003)

[Bur09] BURKE, Bill: *RESTful Java with Jax-RS*. Sebastopol : O'Reilly Media, Inc., 2009

[ces] *Cessna 182Q Skylane Normal Checklists*. `http://www.shorelineflyingclub.com/aircraft/doc/02Nchecklist.pdf`, Abruf: 26.04.2015

[Cha09] CHACON, Scott: *Pro git*. New York City : Apress, 2009

[Cou12] COULOURIS, George F.: *Distributed systems: Concepts and design*. 5th ed. Boston : Addison-Wesley, 2012

[Eic14] EICKELPASCH, Alexander: Funktionaler Strukturwandel in der Industrie: Bedeutung produktionsnaher Dienste nimmt zu. In: *DIW-Wochenbericht* 81 (2014), Nr. 33, S. 759–770

[ENS02] ELMASRI, Ramez A. ; NAVATHE, Shamkant B. ; SHAFIR, Angelika: *Grundlagen von Datenbanksystemen*. 3., überarb. Aufl. München : Pearson-Studium, 2002

[Gei13] GEIGER, Sabrina: *Konzeption und Entwicklung einer auf Smartphones optimierten mobilen Anwendung für kollaboratives Checklisten-Management*, Ulm University, Bachelorarbeit, 2013

[GL02] GILBERT, Seth ; LYNCH, Nancy: Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. In: *ACM SIGACT News* 33 (2002), Nr. 2, S. 51–59

[GL12] GILBERT, Seth ; LYNCH, Nancy A.: Perspectives on the CAP Theorem, Institute of Electrical and Electronics Engineers, 2012

[GR09] GRAY, Jim ; REUTER, Andreas: *Transaction processing: Concepts and techniques*. Posts & Telecom Press and Elsevier, 2009

[HR99] HÄRDER, Theo ; RAHM, Erhard: *Datenbanksysteme: Konzepte und Techniken der Implementierung*. Bd. 2. Heidelberg : Springer, 1999

[ICA14] ICAO: Safety Report. (2014). http://www.icao.int/safety/Documents/ICAO_2014%20Safety%20Report_final_02042014_web.pdf, Abruf: 26.04.2015

[jav] *Java EE 7 Technologies*. http://www.oracle.com/technetwork/java/javaee/tech/index-jsp-142185.html, Abruf: 26.04.2015

[Jen06] JENDROCK, Eric: *The Java EE 5 Tutorial*. Upper Saddle River : Prentice Hall Professional, 2006

[jer] *RESTful Web Services in Java*. https://jersey.java.net/, Abruf: 26.04.2015

[jir] *JIRA*. https://de.atlassian.com/software/jira, Abruf: 26.04.2015

[Joh] JOHANSEN, Anatol: *70 Prozent aller Flugzeugabstürze ließen sich vermeiden*. http://www.welt.de/print-welt/article162064/70-Prozent-aller-Flugzeugabstuerze-liessen-sich-vermeiden.html, Abruf: 26.04.2015

[jqu] *JQuery Mobile*. https://jquerymobile.com/, Abruf: 26.04.2015

[Kay08] KAYAL, Dhrubojyoti: *Pro Java EE Spring Patterns: Best Practices and Design Strategies Implementing Java EE Patterns with the Spring Framework*. New York City : Apress, 2008

[KE11] KEMPER, Alfons ; EICKLER, André: *Datenbanksysteme: Eine Einführung*. Oldenbourg Verlag, 2011

[Köl13] KÖLL, Andreas: *Konzeption und Enwicklung einer auf Tablets optimierten mobilen Anwendung für kollaboratives Checklisten-Management*, Ulm University, Bachelorarbeit, 2013

[KP88] KRASNER, Glenn E. ; POPE, Stephen T.: A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. In: *Journal of object oriented programming* 1 (1988), Nr. 3, S. 26–49

[KS09] KOWALEWSKI, Julia ; STILLER, Silvia: Strukturwandel im deutschen verarbeitenden Gewerbe. In: *Wirtschaftsdienst* 89 (2009), Nr. 8, S. 548–555

[Kun79] KUNG, H. T.: *On Optimistic Methods for Concurrency Control*. Pittsburgh : Carnegie-Mellon University Dept. of Computer Science, 1979

[Lan12] LANGER, David: *MEDo: Mobile Technik und Prozessmanagement zur Optimierung des Aufgabenmanagements im Kontext der klinischen Visite*, University of Ulm, Diplomarbeit, 2012

[leh11] *Verfahrensanweisung für die Kontrolle von Fleischbe- und -verarbeitungsbetrieben im Rahmen der Lebensmittelhygiene-EinzelhandelsVO*. `http://www.lebensmittelinspektion.com/app/download/8682702398/teil+2-cl-k+0232.03.pdf`. Version: 10 2011, Abruf: 26.04.2015

[MDM13] MANSOURI, Najme ; DASTGHAIBYFARD, Gholam H. ; MANSOURI, Ehsan: Combination of Data Replication and Scheduling Algorithm for Improving Data Availability in Data Grids. In: *Journal of Network and Computer Applications* 36 (2013), Nr. 2, S. 711–722

[MKR13] MUNDBROD, Nicolas ; KOLB, Jens ; REICHERT, Manfred: Towards a System Support of Collaborative Knowledge Work. In: *Business Process Management Workshops* Springer, 2013, S. 31–42

[MR14] MUNDBROD, Nicolas ; REICHERT, Manfred: Process-Aware Task Management Support for Knowledge-Intensive Business Processes: Findings, Challenges, Requirements. In: *Enterprise Distributed Object Computing Conference Workshops and Demonstrations (EDOCW), 2014 IEEE 18th International* IEEE, 2014, S. 116–125

[MyS] MYSQL, AB: *MySQL*. `http://www.mysql.com/`, Abruf: 26.04.2015

[Paw98] PAWLOWSKY, Peter: Integratives Wissensmanagement. In: *Wissensmanagement*. Springer, 1998, S. 9–45

[pho] *PhoneGap*. `http://phonegap.com/`, Abruf: 26.04.2015

[PL94] PAHLAVAN, Kaveh ; LEVESQUE, Allen H.: Wireless Data Communications. In: *Proceedings of the IEEE* 82 (1994), Nr. 9, S. 1398–1430

[PMLR13] PRYSS, Rüdiger ; MUNDBROD, Nicolas ; LANGER, David ; REICHERT, Manfred: Supporting medical ward rounds through mobile task and process management. In: *Information Systems and e-Business Management* (2013), Nr. 1, S. 107–146

[pro] *proCollab*. `http://proCollab.de`, Abruf: 26.04.2015

[PS12] PFIFFNER, Martin ; STADELMANN, Peter: *Wissen Wirksam Machen: Wie Kopfarbeiter Produktiv Werden*. Frankfurt am Main : Campus Verlag, 2012

[Rei13] REICH, Daniel: *Konzeption und Entwicklung eines Cloud-basierten Persistenz-Systems für kollaboratives Checklisten-Management*, Ulm University, Bachelorarbeit, 2013

[res] *Restlet*. `http://restlet.com/`, Abruf: 26.04.2015

[RFG13] REMPEL, Andreas ; FRIEDRICH, Kai ; GROSS, Andreas: *Optimierung eines Workflows für Softwareentwickler bei der Bearbeitung von Arbeitspaketen*, Universität Stuttgart, Studienarbeit, 2013

[Thi13] THIEL, Norman: *Konzeption und Entwicklung einer Web-Applikation für kollaboratives Checklisten-Management*, Ulm University, Bachelorarbeit, 2013

[Til09] TILKOV, Stefan: REST und HTTP. In: *Einsatz der Architektur des Web für Integrationsszenarien, dpunkt. verlag* (2009)

[Tv07] TANENBAUM, Andrew S. ; VAN STEEN, Maarten: *Verteilte Systeme: Grundlagen und Paradigmen*. 2., aktualisierte Aufl. München and Boston : Pearson Studium, 2007

[Tv08] TANENBAUM, Andrew S. ; VAN STEEN, Maarten: *Verteilte Systeme: Prinzipien und Paradigmen*. 2., aktualisierte Aufl. München : Pearson Studium, 2008

[Vog07] VOGELS, Werner: Amazons Dynamo. In: *All Things Distributed* (2007)

[WC98] WESSELS, Duane ; CLAFFY, Kim: ICP and the Squid web cache. In: *Selected Areas in Communications, IEEE Journal on* 16 (1998), Nr. 3, S. 345–357

[wil] *WildFly*. `http://wildfly.org/`, Abruf: 26.04.2015

[win] *Apache Wink*. `https://wink.apache.org/`, Abruf: 26.04.2015

[Zie13] ZIEGLER, Jule: *Konzeption und Entwicklung eines Cloud-basierten Servers für kollaboratives Checklisten-Management*, Ulm University, Bachelorarbeit, 2013

Name: Enrico Weigelt Matrikelnummer: 731238

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Enrico Weigelt