Modellgetriebene homogenisierte Datenhaltung und Synchronisation mit vorhersagbaren Eigenschaften für plattformübergreifende Anwendungen

Förderkennzeichen: 01 S12012

AP 2

Anwendungsszenarios und Anforderungsanalyse

Autoren: Eric Gailus (B2M), Benjamin Glatt (FZI), Gerald Hübsch (CAS) , Klaus Krogmann (FZI), Antonia Schwichtenberg (CAS)

Fertiggestellt am: 30.09.2012

Schlagworte: Modellierung, Datenhaltung

Projektpartner

GEFÖRDERT VOM

Bundesministerium für Bildung und Forschung

Änderungshistorie

Version	Änderungen von	Datum	Anmerkung
1.0	Alle	28.09.2012	Initiale Fassung

ToDos

Abschnitt	von	Beschreibung	Pritorität	Todo

Inhalt

Inhalt **3**

Abbildungen **4**

1 Einleitung **1**

2 Anwendungsszenarios **2**

2.1 Offlinefähiges xRM 3

2.1.1 Motivation 3

2.1.2 Beschreibung der Funktionalität aus Benutzersicht 4

2.1.3 Fazit 9

2.2 Multiplattformlösung für mobile Endgeräte 12

2.2.1 Motivation 12

2.2.2 Szenario „Hier bin Ich“ – Lokalisierungsdienst 12

3 Anforderungsanalyse **15**

3.1 Datenschichtmodellierung und domänenspezifische Sprache 15

3.1.1 Modellierung von Entitäten und Beziehungen 15

3.1.2 Modellierung von Dateneigenschaften und -Behandlung 15

3.1.3 Anforderungsübersicht 16

3.2 Toolunterstützung 17

3.3 Homogene Datenzugriffs-API 17

3.4 Laufzeitmechanismen 18

3.4.1 Verteilung und Kommunikation 18

3.4.2 Basisfunktionalität des Applikationsservers 19

3.4.3 Variabilität des Datenmodells 19

3.4.4 Versionierung von Datensätzen 19

3.4.5 Clientseitiges Caching 19

3.4.6 Datensynchronisation 20

3.4.7 Prefetching und Offline-Management von Daten 21

3.4.8 Anforderungsübersicht 21

3.5 Anbindung von Bestandstechnik 23

3.6 Nicht-technische Anforderungen 24

4 Stand der Technik im Umfeld der Anforderungen 24

4.1 Plattformübergreifende Datenmodellierung 24

4.1.1 Implementierung 24

4.1.2 Datenbeschreibung 24

4.1.3 Datenmodellierung 26

4.1.4 Transformation zwischen den Abstraktionsschichten 26

4.2 Laufzeitmechanismen 27

4.2.1 Persistenzschichten für Serverumgebungen 27

4.2.2 Persistenzschichten für mobile Plattformen 30

4.2.3 Datensynchronisation und Konfliktlösungsstrategien 35

4.2.4 Caching 41

5 Literaturverzeichnis 43

Abbildungen

Abbildung 1: Anwendungsfalldiagramm – „Hier Bin Ich“ Dienst 13

Abbildung 2- Abstraktionsebenen der Datenmodellierung 24

1 Einleitung

Mobile Kommunikation, Internetdienste und mobile Anwendungen sind zu einem zentralen Bestandteil unserer modernen Informations- und Kommunikationsgesellschaft geworden. Bis zum Jahr 2013 soll sich die Zahl der Nutzer von Smartphone Applikationen weltweit von rund 300 Millionen in diesem Jahr auf 975 Millionen mehr als verdreifachen. [1] Daher wächst auch die Erwartung an Unternehmen ihr Produktportfolio um mobile Anwendungen zu erweitern um dadurch ihre Kundenbindung zu erhöhen oder Neukunden zu gewinnen, bzw. neue Geschäftsfelder zu erschließen.

Wenn in diesem Kontext über mobile Anwendungen diskutiert wird, sind streng genommen zwei Ausprägungen zu unterscheiden. Zum einen gibt es die „Apps“, kleine Softwareprogramme für mobile Endgeräte, die für die entsprechende Plattform angepasste Anwendungen darstellen. Zum anderen mobile Dienste, welche häufig ebenfalls unter dem Überbegriff mobiler Anwendungen zusammengefasst werden, auf die allerdings über einen Webbrowser zugegriffen wird. Diese sind somit keine eigenständigen, für die jeweiligen Plattformen angepasste, Applikationen. Die Nachteile die sich daraus ergeben sind unter anderem das Fehlen nativer Bedienelemente und eine sehr eingeschränkte Nutzung der Funktionalitäten der zugrunde liegenden Plattformen. Dies führt zu einer geringen Akzeptanz browserbasierter Apps bei Endbenutzern, wie eine Umfrage unter 500 Experten des „Bundesverband Informationswirtschaft, Telekommunikation und neue Medien e.V.“ ergab. „Fast drei Viertel der Befragten sind der Meinung, dass sich Apps in den kommenden fünf Jahren bei Verbrauchern breit durchsetzen. Nur 22 Prozent sehen mobile Websites und sonstige browserbasierte Anwendungen vorne.“ [2]

Die Schwierigkeit hierbei ist die Tatsache, dass für jede mobile Plattform andere Technologien und API’s zur Entwicklung der Anwendungen und zur Nutzung der entsprechenden plattformspezifischen Funktionalitäten einzusetzen sind. Dies ist der Fall, da keine befriedigende allgemein akzeptierte technische Basis für alle Plattformen zur Verfügung steht, was den Entwicklungsaufwand bzw. den Aufwand der Portierung einer App für mehrere Plattformen ungemein erhöht.

Im Rahmen des MOHITO-Projekts sollen vor diesem Hintergrund die Datenhaltung und -synchronisation plattformübergreifend durch den Einsatz modellgetriebener Techniken homogenisiert werden. Die dadurch aus einem zentralen Beschreibungsmodell erzeugbaren Datenhaltungsschichten können in ihrer Qualität (Ressourcenbedarf, Performanz, Verfügbarkeit) abgeschätzt werden. Als Ziel werden plattform- und applikationsspezifische Datenhaltungsschichten und -Frameworks sowie Synchronisationsmechanismen aus einer applikationsspezifischen Modellbeschreibung erzeugt, die durch eine zielgerichtete Migration auch

in Bestandssoftware integriert werden können. Der manuelle Implementierungsaufwand für neue Plattformen und neue Features sinkt damit signifikant.

Das vorliegende Dokument wurde im Rahmen des Arbeitspaketes 2 "Anforderungsszenarios und Anforderungsanalyse" des MOHITO Projektes erstellt und beschreibt zwei Anwendungsszenarios, sowie die sich daraus ergebenden Anforderungen an die zu entwickelnde MOHITO Plattform. Die im Dokument genannten Anforderungen sind in Muss-, Soll- und Kann-Anforderungen kategorisiert. Muss-Anforderungen definieren den Kern des MOHITO Projekts. Sie müssen am Projektende für einen erfolgreichen Projektabschluss umgesetzt sein. Die Umsetzung von Soll-Anforderungen ist wünschenswert, jedoch nicht projektentscheidend. Sie wird von den Projektpartnern angestrebt, sofern die Erfüllung der Muss-Anforderungen dadurch nicht gefährdet wird. Die Umsetzung von Kann-Anforderungen ist ebenfalls nicht projektentscheidend. Ihre Umsetzung ist optional und den Soll-Anforderungen nachgeordnet.

Im folgenden Kapitel werden zwei Anwendungsszenarios beschrieben: 1. Eine mobile, offline-fähige xRM-Anwendung und 2. Ein Lokalisierungsdienst. Beide Szenarios adressieren die zentralen Anforderungen an die zu entwickelnde MOHITO Plattform, unterscheiden sich aber im zu Grunde liegenden technischen Setup.

In Kapitel 3 werden die sich daraus ergebenden Anforderungen genau definiert. Der Fokus hierbei liegt auf den technischen Anforderungen an die Datenschichtmodellierung und den Datenzugriff, es werden aber auch Anforderungen an die zu leistende Toolunterstützung, an Laufzeitmechanismen, sowie nicht-technische Anforderungen beschrieben.

In Kapitel 4 werden die aktuell existierenden, technischen Voraussetzungen beschrieben und der Stand der Technik in diesem Bereich skizziert.

2 Anwendungsszenarios

Im Folgenden werden zwei Anwendungsszenarien beschrieben: Eine mobile, offline-fähige xRM Applikation (Kapitel 2.1) und ein "Hier-bin-ich" Lokalisierungsdienst (Kapitel 2.2) Anhand der konkreten Beschreibung des Szenarios werden die Anforderungen, die ein Benutzer an eine mobile Anwendung stellt, spezifiziert.

Die in den einzelnen Kapiteln beschriebenen Anforderungen sind nummeriert [A1.1-1…An] und werden am Ende eines jeden Kapitels tabellarisch zusammengefasst. Dies ermöglicht eine eindeutige Bezugnahme während der gesamten Projektlaufzeit sowie in verschiedenen Tools wie z.B. dem Issue-Tracking Tool Jira.

2.1 Offlinefähiges xRM

Im Folgenden wird ein Szenario für den mobilen Einsatz eines Relationship Management (xRM) Systems beschrieben. Ein xRM System ist eine verallgemeinerte Form eines CRM (Customer Relationship Management) Systems insofern als dass nicht nur die Beziehungen zu Kunden, sondern beispielsweise auch zu Bewerbern, zu Mitarbeitern oder Lieferanten gepflegt werden. Es dient der Verwaltung und Analyse sämtlicher Kontaktinformationen, inklusive der damit verknüpften Informationen wie Termine, Aufgaben, Dokumente, Verkaufschancen, Kampagnen etc.

Es wird gezeigt, dass die offline-Funktionalität und der (automatische) Datenabgleich ein zentrales Feature einer mobilen xRM Applikation sein sollte und wie diese für den Benutzer zur Verfügung gestellt werden sollten. Zwar bestehen bereits Synchronisationsmechanismen, die es beispielsweise ermöglichen, Termine aus der xRM Anwendung in den Kalender des Smartphone zu übernehmen oder Kontakte mit der Kontaktverwaltung des Smartphones zu synchronisieren, der entscheidende Nachteil hierbei ist aber, dass die mit einem Termin oder Kontakt verknüpften Dokumente/Daten dann nicht offline zur Verfügung stehen – da die systemeigenen Tools die Verwaltung dieser verknüpften Zusatzinformationen nicht ermöglichen.

2.1.1 Motivation

Da ein xRM System der Verwaltung sämtlicher Kontaktinformationen dient, liegt es auf der Hand, dass gerade ein solches System auf mobilen Endgeräten zur Verfügung stehen muss. So möchte ein Außendienstmitarbeiter die Daten zu einem Kunden und alle für den Kunden relevanten Dokumente und Termine auf seinen mobilen Endgeräten vor Ort zur Verfügung haben und zwar auch dann wenn keine Internetverbindung zur Verfügung steht.

Zum einen gibt es auf dem Markt der Tablet-PCs und Laptops immer noch viele Geräte, die nicht mit internetfähigem Mobilfunk wie UMTS ausgestattet sind und die daher nur unter Verwendung von WLAN internetfähig sind. Zum anderen muss aber auch bei der Nutzung von mobilfunkbasierten Datendiensten wie UMTS mit Unterbrechungen der Internetverbindung gerechnet werden.

Generell muss zwischen kurzfristigen und langfristigen Unterbrechungen der Internetverbindung unterschieden werden; ihnen liegen typischerweise unterschiedliche Ursachen zugrunde und es resultieren unterschiedliche Anforderungen aus Nutzer-Sicht.

Kurzzeitige Verbindungsunterbrechung

- **Unzuverlässige Netzwerkanbindung im Außendienst:** Häufig treten auf dem Weg zum Kunden kurzzeitige Verbindungsunterbrechungen auf, z.B. verursacht durch Störungen oder während der Fahrt. Der Wechsel von einem Netz in ein anderes ist oftmals auch mit einer Änderung der Bandbreite des verfügbaren Netzes verbunden.

- **Wechselnde Bandbreite durch variierende QoS (Quality of Service) mobiler Funknetze (GSM/UMTS, HSPA/WLAN):** Da die Abdeckung von breitbandigen Netzen wie HSPA+ (21Mbit/s) noch nicht sehr weit ist, stellt sich die Herausforderung, mit Netzen geringer Bandbreite umzugehen. So wäre im Falle eines schmalbandigen Netzes wie GPRS mit 55 kbit/s oder EDGE mit 220 kbit/s eine Priorisierung der zu synchronisierenden Daten erwünscht (wie in Kap 2.1.2.2. gezeigt wird).

Langzeitige Verbindungsunterbrechung

- **Internationale Abwicklung:** Gerade bei Geschäften, die international abgewickelt werden, stellt das Roaming, d.h. der Übergang von einem Netz eines Landes in ein Netz eines anderen Landes eine gewisse Herausforderung dar. Prinzipiell sind zwar mittlerweile alle Smartphones roaming-fähig, doch können die entstehenden Kosten ein Hinderungsgrund für die Nutzung sein oder aber es kann auch abhängig vom Gebiet gar kein Netz zur Verfügung stehen.
- **Policy-bedingt:** Aufgrund der Sicherheitsrichtlinien eines Unternehmens kann generell die Nutzung fremder Netze unterbunden sein, was die offline-Fähigkeit der Applikation unabdingbar macht. Oder aber es kann beim Kunden unterbunden sein, dass sich fremde Rechner mit dem unternehmenseigenen Netz verbinden. In diesem Fall wäre die Nutzung von Mobilfunknetzen eine weiterhin bestehende Möglichkeit, bei der aber nicht nur mit kurzzeitigen Verbindungsunterbrechungen, sondern auch mit langzeitigen Verbindungsunterbrechungen gerechnet werden muss.
- **Verbindungsabbruch beim Kunden:** Abhängig vom Standort oder dem Gebäude des Kunden kann eine Nutzung des Mobilfunknetzes auch für lange Zeit nicht möglich sein. Auch ein Zusammenbruch des WLAN Netzes des Kunden kann eine Ursache für einen Verbindungsabbruch beim Kunden sein.

Die Beispiele zeigen, dass es viele Gründe geben kann, wegen derer das mobile Endgerät keine Internetverbindung hat. Das Arbeiten mit (Teilen) der xRM Anwendung sollte sowohl unter kurzzeitigen als auch unter langzeitigen Verbindungsunterbrechungen möglich sein [A2.1.1-1].

2.1.2 Beschreibung der Funktionalität aus Benutzersicht

Im Folgenden wird beschrieben, wie eine intuitive Benutzung einer mobilen xRM Anwendung aussehen könnte. Folgende Leitlinien sollen hierbei gelten: Einfachheit, Bequemlichkeit und Eleganz. Der Benutzer soll möglichst wenig per Hand machen müssen, sondern stattdessen einmal das Verhalten des Systems konfigurieren können und dann immer automatisch alle benötigten Daten (auch offline) zur Verfügung haben. [A2.1.2-1] Generell darf eine Verbindungsunterbrechung nie zu einem Datenverlust führen [A2.1.2-2].

Da es verschiedene Betriebssysteme gibt, die sich auf dem Markt der mobilen Endgeräte etabliert haben, sollte die xRM Anwendung zumindest auf den gängigen Systemen wie Android und iOS nutzbar sein (in Kapitel 2.2.1 wird näher auf die Herausforderungen der plattformunabhängigen Entwicklung eingegangen). [A2.1.2-3] Die Unterstützung weiterer Betriebssysteme soll prinzipiell möglich sein, ist aber keine Kernanforderung an den innerhalb des Projektes zu entwickelnden Prototyp.

2.1.2.1 Offline-Funktionalität

Angenommen, ein Außendienstarbeiter ist auf dem Weg zum Kunden: Was erwartet er von der Anwendung? Wie will er mit der Applikation auf seinem Smartphone oder Tablett-PC arbeiten?

Die xRM Applikation sollte auch dann gestartet werden können, wenn keine Internetverbindung besteht. [A2.1.2-4] Dabei ist es nicht erforderlich, dass die gesamte Funktionalität zur Verfügung steht. [A2.1.2-5] Häufig beschränkt der Formfaktor eines Gerätes die Darstellungs- und Editiermöglichkeiten; unter Umständen beschränkt sich die Nutzung sogar auf eine Consumer-Funktionalität, bei der man die Daten nur lesen und nicht editieren kann. [A2.1.2-6]

Die Applikation erlaubt eine einfache und intuitive Navigation durch die bestehenden Kontakte und stellt verschiedene vordefinierte Filter zur Verfügung, wie z.B. nur "meine Kontakte", nur "aktuelle Kontakte" oder "alle Organisationen". [A2.1.2-7]

Automatisch alle relevanten Daten offline halten

Generell sollten alle Kontakte (Telefonnummer und Adresse der Ansprechpartner) des Nutzers immer auch offline zur Verfügung stehen [A2.1.2-8]. Beispielsweise möchte der Außendienstmitarbeiter die Adresse des Kunden einem Taxifahrer mitteilen können oder er möchte den Kunden anrufen können, falls er sich verspätet.

Ebenso sollten immer alle Termine des Nutzers offline zur Verfügung stehen. [A2.1.2-9] Die Anzahl der offline verfügbaren Datensätze wird jedoch generell durch die Ressourcen des mobilen Clients begrenzt. Überschreitet ihre Anzahl eine Grenze *n*, sollen nur jeweils die für den Nutzer relevantesten Datensätze, beispielsweise die 500 wichtigsten Termine, zur Verfügung stehen. [A2.1.2-10]

Die Mechanismen zur Feststellung, welche Datensätze die wichtigsten sind, sollten durch das System bereitgestellt werden und sich an der Relevanz für den Nutzer orientieren. [A2.1.2-11] Die Mechanismen zur Relevanzermittlung sollten durch den Nutzer konfigurierbar sein. [A2.1.2-12] Beispielsweise kann mit einfließen, wann die Daten zuletzt bearbeitet wurden oder wann ein Termin stattfindet.

Transparentes Prefetching

Um sich auf den Kundenbesuch vorzubereiten, sollte der Benutzer möglichst wenige Informationen von Hand herunterladen müssen. Beispielsweise sollten alle Termine der nächsten Woche inklusive der damit verknüpften Dokumente, Verkaufschancen etc. offline zur Verfügung stehen. [A2.1.2-13]

Manuelles Herunterladen

Zusätzlich soll der Benutzer auch in der Lage sein, gezielt einzelne Daten (z.B. ein bestimmtes Dokument) [A2.1.2-14] oder die gesamte Akte (Dokumente, Vorgänge, Verkaufschancen, Kampagnen etc.) eines Kontaktes oder Termins [A2.1.2-15] offline verfügbar zu machen. Auch ein gezieltes Herunterladen einzelner Typen von in der Akte gespeicherten Informationen sollte möglich sein, so dass man z.B. alle Dokumente zu einem Kontakt herunterladen kann, nicht aber die Verkaufschancen etc. [A2.1.2-16].

Sinnvoll wäre auch ein vereinfachtes Herunterladen aller Dokumente, die mit einer bestimmten Organisation verknüpft sind, so dass man nicht für jeden Kontakt (zu der Organisation) die Dokumente/Akten einzeln herunterladen muss. [A2.1.2-17] Da die Dokumente, die mit einem Kontakt verknüpft sind, weiter mit anderen Informationen verknüpft sein können, stellt sich die Frage, wie weit man an dieser Stelle traversieren möchte. Für den Benutzer intuitiv wäre hier standardmäßig nur die direkten (first-level) Verknüpfungen herunterzuladen; für den Fall, dass auch weitere Verknüpfungen (second-level, third-level) geholt werden sollen, könnte die maximale Verknüpfungstiefe vom Nutzer vorgegeben werden [A2.1.2-18].

Im Falle der manuell heruntergeladenen Daten, muss es eine Verdrängungsstrategie geben, die Daten, die nicht mehr gebraucht werden, auf dem mobilen Endgerät löscht. Das Löschen der offline-Daten muss der Benutzer beeinflussen können, da es sein könnte, dass der Benutzer einen bestimmten Datensatz immer zur Verfügung haben will, auch wenn er ihn in den letzten Monaten nicht mehr benutzt hat. [A2.1.2-19] Dies ist im Falle der relevanten Daten, die automatisch offline zur Verfügung gestellt und für den Nutzer transparent verwaltet werden, nicht erforderlich – die nicht mehr relevanten Daten werden durch die aktuell relevanten Daten verdrängt.

Visualisierung

Der Unterschied zwischen heruntergeladen und nicht-heruntergeladen Akten oder Dokumenten sollte immer auf den ersten Blick deutlich sein, z.B. durch Ausgrauen der nicht-heruntergeladen Dokumente [A2.1.2-20]. Im offline-Modus setzt dies voraus, dass die Listen aller Daten immer zur Verfügung stehen. Es sollte also möglich sein, die Liste aller Kontakte oder Projekte zu sehen, wenn auch im offline-Modus nicht alle geöffnet werden können. [A2.1.2-21] Klickt man ein nicht heruntergeladenes Dokument an, so fragt das mobile Endgerät, ob es eine Verbindung zum Internet herstellen soll. [A2.1.2-22]

Ferner sollte es einen Filter geben, der es ermöglicht, nur die heruntergeladen Dokumente anzuzeigen, damit ein langes Navigieren und Suchen nicht erforderlich ist [A2.1.2-23].

2.1.2.2 Synchronisation

Generell gibt es drei Fälle, in denen der Datenabgleich stattfinden kann: 1. Es besteht eine Verbindung zum Internet, 2. es besteht keine Verbindung oder 3. die Verbindung wird gerade hergestellt.

1. Wenn eine Internet-Verbindung besteht, sollten immer alle Daten automatisch synchronisiert werden (echt-Zeit). [A2.1.2-24] Da die Dauer der Verbindung gerade im mobilen Einsatz nicht immer vorhersagbar ist, ist das für die Synchronisation verfügbare Zeitfenster in einigen Fällen unbekannt. Deshalb muss der Synchronisationsmechanismus in der Lage sein, mit plötzlichen Verbindungsabbrüchen umzugehen und dies dem Nutzer entsprechend zu signalisieren.

2. Wenn keine Internet-Verbindung besteht, sollte der Benutzer - sobald er Änderungen an einem Kontakt oder einem Dokument vorgenommen hat - in der Lage sein, einzelne Daten [A2.1.2-25] oder alle geänderten Daten [A2.1.2-26] gezielt manuell zu synchronisieren. Wenn er dies tut (und sein Endgerät ist nicht online ist), sollte dem Benutzer eine Rückfrage präsentiert werden, ob das Gerät eine Internet-Verbindung herstellen soll. [A2.1.2-27] Antwortet der Benutzer mit "Nein", so passiert nichts, antwortet er mit "Ja", so wird die Verbindung hergestellt und auf dem Server geprüft, ob dort etwas an den entsprechenden Daten geändert wurde. Nur wenn es auf dem Server keine Änderungen an diesen Daten gibt, werden die Änderungen übertragen; andernfalls werden die Daten als in Konflikt stehend markiert.

   Es sollte immer auf den ersten Blick sichtbar sein, ob ein Kontakt/Termin/Dokument synchronisiert ist, nicht-synchronisiert ist oder ob ein Konflikt vorliegt [A2.1.2-28]. Aufgrund der Displaygröße eines Smartphones und der Komplexität der hier vorliegenden Daten, ist es nicht erwünscht, Konflikte auf der mobilen Anwendung manuell zu beheben [A2.1.2-29]; die Konfliktbehebung kann eigentlich nur sinnvoll auf einem PC erfolgen [A2.1.2-30]. Dennoch sollte es möglich sein, sich auf dem mobilen Endgerät die andere (Server-) Version der Daten anzeigen zu lassen, damit man die eigene Änderung ggf. verwerfen kann. [A2.1.2-31] Sofern ein automatisches Zusammenführen (Merge) beider Datensätze möglich ist, sollte dies vorgeschlagen werden. Der Benutzer kann dann dem Vorschlag zustimmen oder ihn ablehnen. [A2.1.2-32]

3. Wenn eine Internetverbindung hergestellt wird und die Applikation geöffnet ist, sollten automatisch alle Daten synchronisiert werden [A2.1.2-33] (mit vorheriger Frage an den Benutzer, ob dies geschehen soll und eventuell inklusive einer Benachrichtigung, welche Datensätze synchronisiert werden [A2.1.2-34]).

   Da während der Synchronisation die Internetverbindung abbrechen kann, ist die beschriebene Kennzeichnung wichtig, um zu identifizieren, welche Daten synchronisiert sind, welche nicht und wo Konflikte vorliegen.

Da unter Umständen sehr viele Daten synchronisiert werden müssen, bzw. das zur Synchronisation zur Verfügung stehende Zeitfenster im mobilen Fall unbekannt ist, sollte an dieser Stelle eine bestimmte Priorität der zu synchronisierenden Daten definiert werden. Dies kann entweder im System vordefiniert sein, z.B. datensatztypabhängig (zuerst alle Kontakte, dann alle Dokumente, dann …) oder in Abhängigkeit von der Relevanz, z.B. dem Zugriffszeitpunkt oder der Zugriffshäufigkeit auf die offline-Datensätze.[1] [A2.1.2-35] Dies könnte auch vom Benutzer beeinflusst, bzw. konfiguriert werden. [A2.1.2-36] Eine weitere, elegante Lösung an dieser Stelle wäre es, eine Ausgangsbox zur Verfügung zu stellen, in der alle Daten landen, die offline auf dem Endgerät geändert wurden. [A2.1.2-37] Hier könnte der Benutzer die Reihenfolge der zu synchronisierenden Daten ändern [A2.1.2-38] und gezielt Daten abwählen, die nicht synchronisiert werden sollen (z.B. weil die Bandbreite sehr gering ist und die Synchronisation zu lange dauern würde oder weil die aufkommenden Kosten zu hoch sind). [A2.1.2-39] Die nicht synchronisierten, aber offline geänderten Daten müssen beim erneuten Synchronisationsversuch wieder in der Liste auftauchen.

2.1.2.3 Sicherheit

Da eine xRM Applikation den Zugriff auf sensible Daten ermöglicht, ist auch das Thema Sicherheit hier relevant. Generell gibt es verschiedene Fälle, in denen Sicherheit bedacht werden muss.

1. Verlust des Gerätes
   Gemäß einer vom BITKOM (Bundesverband Informationswirtschaft, Telekommunikation und neue Medien e.V.) beauftragen, repräsentativen Studie hat jeder 10. Deutsche schon mal sein Mobiltelefon verloren.[2] Deshalb sollen:
   a. Im Offline-Cache gespeicherte Daten nicht ohne Authentifizierung gelesen werden können und sie sollten verschlüsselt sein [A2.1.2-40]
   b. In sehr restriktiven Fällen, bei denen ein Maximum an Sicherheit gefordert ist, könnte die Option zur offline-Verfügbarkeit auf einen Teil der gesamten Daten des Unternehmens begrenzt sein. [A2.1.2-41] Prinzipiell ist hier eine Auswahl anhand des Datentyps (Schema-Ebene), einzelner Daten (Instanz-Ebene) oder in Abhängigkeit vom Benutzer (nur seine Daten) möglich.
2. Angriff während der Datenübertragung
   Die Daten sollen für die Übertragung SSL-verschlüsselt sein. [A2.1.2-42]
3. Sicherheit des Servers

[1] Hier könnte der Relevanz-Mechanismus zur Feststellung, welche Dokumente offline gehalten werden, wiederverwendet werden oder aber man könnte auch die sich aus der offline-Nutzung ergebende Relevanz in den serverseitigen Mechanismus mit einfließen lassen.

[2] Vgl. http://www.bitkom.org/de/presse/8477_72651.aspx

Der Server, auf dem die Daten liegen, sollte bestmöglich geschützt sein. [A2.1.2-43]

2.1.3 Fazit

Aus der Art und Weise, wie ein xRM System genutzt wird – nämlich vor allem beim Kunden vor Ort – ergibt sich die Anforderung an die offline-Funktionalität, denn zum einen besteht die Möglichkeit, dass die Internetverbindung kurz abbricht, zum anderen können lange offline-Zeiten unausweichlich sein.

Es wurde gezeigt, dass prinzipiell alle für den Nutzer relevanten Daten immer offline zur Verfügung stehen sollen. Zusätzlich soll es aber auch möglich sein, einzelne Datensätze, aber auch Gruppen von Datensätzen für die offline-Benutzung herunterzuladen. Dann soll sichtbar sein, welche Datensätze offline zur Verfügung stehen und welche nicht. Für die Synchronisation der geänderten Datensätze wurden verschiedene Strategien beschrieben (manuell, automatisch und semi-automatisch mit konfiguriertem Verhalten).

Nummer	Anforderung	Muss	Soll	Kann
	Generell			
A2.1.1-1	Offline Fähigkeit	X		
A2.1.2-1	Automatisch alle relevanten Daten offline		X	
A2.1.2-2	Vermeidung von Datenverlust	X		
A2.1.2-3	Für alle gängigen Betriebssysteme	X		
A2.1.2-4	Start ohne Internetverb. möglich	X		
A2.1.2-5	Gesamte Funktionalität offline verfügbar			X
A2.1.2-6	Offline nur Consumer-Funktion			X
A2.1.2-7	Filter (meine, aktuelle)	X		
	Automatisches Herunterladen relevanter Datensätze (Prefetching)			
A2.1.2-8	Kontakte des Nutzers offline	X		
A2.1.2-9	Termine des Nutzers offline	X		
A2.1.2-10	*n* relevanteste Datensätze offline		X	
A2.1.2-11	Systemgestützte Relevanzermittlung für Datensätze		X	
A2.1.2-12	Relevanzbestimmung konfigurierbar			X
A2.1.2-13	Prefetching der Verknüpfungen aller Termine der nächsten Woche		X	
	Manuelles Herunterladen			

A2.1.2-14	Herunterladen einzelner Daten	X		
A2.1.2-15	Herunterladen gesamte Akten		X	
A2.1.2-16	Herunterladen von Typen von Daten			X
A2.1.2-17	Herunterladen verknüpfter Daten	X		
A2.1.2-18	Herunterladen aller Verknüpfungen mit konfigurierbarer Verknüpfungstiefe			X
A2.1.2-19	Beeinflussen der Verdrängungsstrategie manuell heruntergeladener Daten		X	
A2.1.2-22	Nachfrage "Verbindung herstellen" beim Öffnen von nicht-heruntergeladenen		X	
	Visualisierung			
A2.1.2-20	Direkte Sichtbarkeit welche Daten heruntergeladen wurden	X		
A2.1.2-21	Offline alle Server-Daten als Liste		X	
A2.1.2-23	Filter nur heruntergeladene Daten		X	
A2.1.2-28.	Direkte Sichtbarkeit welche Daten synchronisiert sind, nicht synchronisiert oder im Konflikt	X		
A2.1.2-37	Filter nur nicht synchronisierte Daten (Ausgangsbox)		X	
	Synchronisation			
A2.1.2-24	Automatisches Sync. im online Modus	X		
A2.1.2-25	Manuelles Sync. einzelner Daten	X		
A2.1.2-26	Manuelles Sync. aller Daten	X		
A2.1.2-27	Nachfrage "Verbindung herstellen?" beim manuellen Sync. im offline-Modus		X	
A2.1.2-33	Automatisches Sync. aller Daten	X		
A2.1.2-34	Nachfrage ob synchronisiert werden soll und Benachrichtigung welche (beim automatischen Sync.)	X		
A2.1.2-35	Priorität der zu synchronisierenden Daten		X	
A2.1.2-36	Priorität konfigurierbar		X	
A2.1.2-38	Reihenfolge der zu synchronisierenden Daten explizit angeben		X	
A2.1.2-39	Abwählen einzelner zu synchronisierender Daten		X	
A2.1.2-29	Konfliktbehebung auf Smartphone			X
A2.1.2-30	Konfliktbehebung auf PC	X		

A2.1.2-31	Anzeige der Server-Version der konfliktiven Daten			X
A2.1.2-32	Vorschlag zum automatischen Merge			X
	Sicherheit			
A2.1.2-40	Lesen der offline Daten nicht möglich		X	
A2.1.2-41	Beschränkung der Option zur offline-Verfügbarkeit (nicht alle Unternehmensdaten)			X
A2.1.2-42	Verschlüsselte Datenübertragung	X		
A2.1.2-43	Sicherheit des Servers	X		

2.2 Multiplattformlösung für mobile Endgeräte

Im Folgenden soll die Bandbreite mobiler Anwendungsszenarien detaillierter betrachtet und die daraus resultierende Notwendigkeit einer Multiplattformlösung für mobile Endgeräte näher beleuchtet werden.

2.2.1 Motivation

Vor dem Hintergrund der in der Einleitung geschilderten uneinheitlichen technischen Basis sind die geplante Entwicklung eines Multiplattform-Framework und entsprechender Generatoren für Datenhaltung und Datensynchronisation zu sehen, welche eine plattformübergreifende homogene Datenhaltung gewährleisten sollen. Die Lösung soll hierbei speziell die Bedürfnisse verteilter, mobiler Anwendungen berücksichtigen. Des Weiteren ist geplant, durch den Einsatz eines „Mobile Mediation Layer“, die Anbindung existierender Internetdienste bzw. browserbasierter Dienste an native mobile Anwendungen zu vereinfachen. Welche Anforderungen sich an solch eine Multiplattformlösung ergeben, wird nachfolgend exemplarisch anhand der Beschreibung eines Dienstszenarios verdeutlicht.

2.2.2 Szenario „Hier bin Ich“ – Lokalisierungsdienst

Die grundlegende Idee zu diesem Dienst bzw. der dahinterstehenden App besteht darin, einen Freund, Bekannten oder Geschäftspartner an einem sehr belebten, öffentlichen Ort zu finden. Solch ein Ort kann ein Bahnhof oder Flughafen sein, an dem eine Person erwartet wird, aber auch ein Volksfest, oder sonstige Massenveranstaltung, auf der sich Menschen finden wollen. Dies stellt Aufgrund der Menschenmassen oft ein größeres Problem dar, für das es keine echte Hilfestellung durch vorhandene Apps gibt. Sicherlich gibt es Anwendungen auf den einschlägigen mobilen Plattformen wie „Latitude“ unter Android, oder „Find my Friends“ für das iPhone, welche den Aufenthaltsort einer Person auf einer Karte anzeigen können. Diese bieten aber eine recht ungenaue Lokalisierung, welche in den geschilderten Szenarien wenig hilfreich ist.

Die Idee ist es hier eine Möglichkeit zu schaffen die eigene Position über die Ortsangabe hinaus mit Zusatzinformationen anzureichern, welche das Auffinden erleichtern. Denkbar ist hier beispielsweise das Verbinden der Position auf einer Karte, die den ungefähren Standort angibt, mit einem, mit dem Smartphone geschossenen und hochgeladenen Foto eines markanten Objekts in der Umgebung und der textuellen Beschreibung der relativen Position zu diesem. Darüber hinaus wäre es möglich den Standort mit Informationen anderer Dienste wie Wetter, Verkehr oder „Point of Interests“ anzureichern.

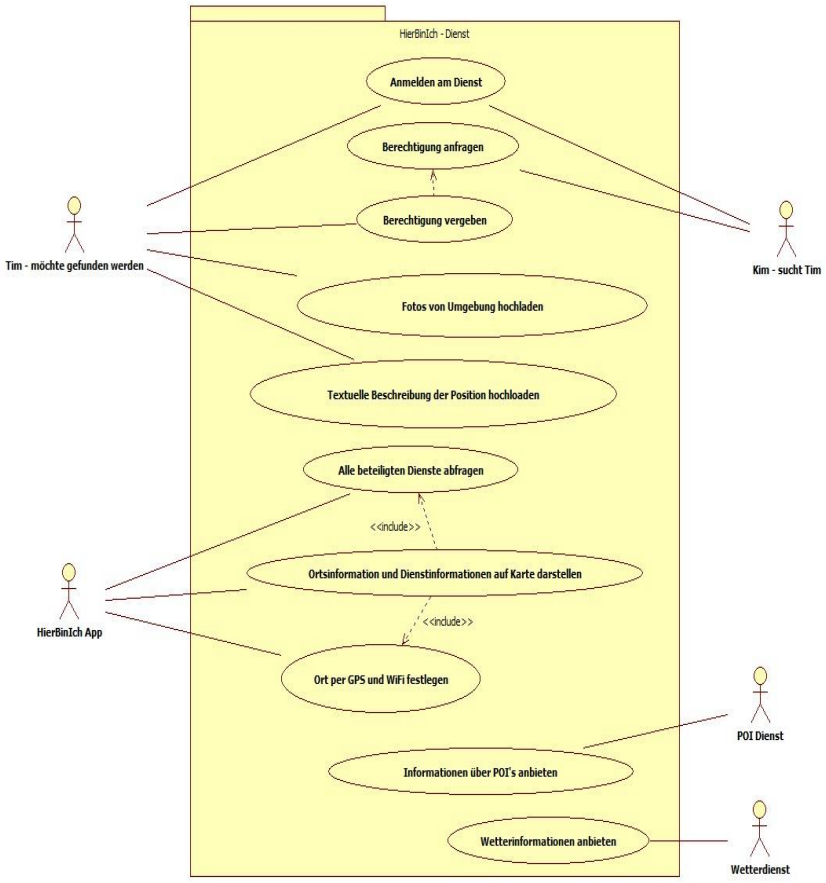

Abbildung 1: Anwendungsfalldiagramm – „Hier Bin Ich“ Dienst

Eine Anwendung, die diese Funktionalitäten zur Verfügung stellen soll, muss eine Anbindung an mehrere beteiligte Dienste gewährleisten. Darüber hinaus soll es möglich sein, Dienste innerhalb einer Servicekategorie gegeneinander auszutauschen. Über den MML als Mediator ist diese Möglichkeit grundsätzlich gegeben, da Clientanwendungen ein generisches Dienstinterface innerhalb einer Servicekategorie zur Verfügung gestellt wird, hinter welchem sich serverseitig unterschiedliche Dienstanbieter verbergen können. So können beispielsweise Daten für einen Wetterdienst von unterschiedlichen, austauschbaren Wetter Anbietern stammen. Welcher Dienstanbieter beim Bedienen einer Clientanfrage zum Zug kommt, wird vom MML unter Beachtung von Verfügbarkeit, Performance, Kosten oder Benutzervorgaben entschieden.

Des Weiteren müssen kurze Netzausfälle und längere Offlinephasen behandelt werden, welche die Aktualität bzw. Qualität der Daten beeinflussen. Nicht mehr aktuelle und daher in ihrer Qualität geminderte Daten müssen entsprechend visualisiert werden. In dem beschriebenen Anwendungsfall würde das bedeuten, dass beispielsweise bei einem Netzausfall die aktuelle Position der zu findenden Person nicht mehr zum Dienst gemeldet werden kann.

Nutzern dieser Information, muss diese Situation signalisiert werden. Gleiches gilt natürlich auch umgekehrt für den Fall, dass die Verbindung zum Dienst wieder besteht und aktualisierte Informationen vorliegen. In diesem Fall müssen auch die lokal auf dem mobilen Client zwischengespeicherten Daten mit dem Server abgeglichen werden. Komm es hierbei zu Synchronisationskonflikten, können diese durch Prioritätsverfahren über definierte Regeln automatisiert aufgelöst werden.

Bei der Berücksichtigung der Erfordernisse eines mobilen Clients ist auch auf die zu übertragende Datenmenge bzw. Bandbreite zu achten, gerade bei dem Versuch möglichst viele Informationen unterschiedlichster Dienste zu verknüpfen und aktuell zu halten. Hier ist es sinnvoll, Informationen nach dem Grad ihres Nutzens und der dafür benötigten Aktualisierungsrate zu kategorisieren. Anhand dieser Kategorien können unterschiedliche Datenhalbwertszeiten definiert werden, mit denen beispielsweise Caching- und Synchronisationsverfahren optimiert werden können.

	Anforderung	**Muss**	**Soll**	**Kann**
A2.2-1	Offline Fähigkeit	X		
A2.2-2	Dienstprofile und Nutzerdaten müssen immer verfügbar sein	X		
A2.2-3	Nicht mehr aktuelle Daten müssen entsprechend visualisiert werden		X	
A2.2-4	Anbindung an mehrere beteiligte Dienste	X		
A2.2-5	Bandbreitenlimitierung mobiler Geräte muss berücksichtigt werden (Caching)		X	
A2.2-6	Mechanismus zur Reduktion des Datenvolumens		X	
A2.2-7	Automatische Konfliktbehebung	X		
A2.2-8	Manuelle Konfliktbehebung auf Smartphone			X

3 Anforderungsanalyse

In diesem Kapitel werden die technischen Anforderungen an Datenschichtmodellierung und den Datenzugriff dokumentiert. Weiterhin werden Anforderungen an die zu realisierende Toolunterstützung, an Laufzeitmechanismen, sowie nicht-technische Anforderungen beschrieben.

3.1 Datenschichtmodellierung und domänenspezifische Sprache

Neben der klassischen Datenmodellierung besteht die Herausforderung im Rahmen des MOHITO Projektes vor allem in der Definition von funktionalen und nicht-funktionalen Eigenschaften der Daten und deren Behandlung durch die Datenschicht. Letzteres wird von heutigen Modellierungswerkzeugen nicht unterstützt und stellt einen der Kernbeiträge des Projektes dar.

3.1.1 Modellierung von Entitäten und Beziehungen

Unter der „klassischen Datenmodellierung“ ist die Definition von Entitäten samt ihrer Attribute und den zwischen ihnen existierenden Beziehungen zu verstehen. Hierbei gilt es sowohl Basisdatentypen, wie Zeichenketten, oder numerische Werte, als auch selbst definierte, komplexere Datentypen als Attribute zu zulassen. Um eine ausreichende Flexibilität zu ermöglichen, sind darüber hinaus auch Attribute und Referenzen unterschiedlicher Kardinalität notwendig.

Ziel der Modellierung ist ein zentrales Datenmodell, das sowohl auf den clientseitigen, als auch auf den serverseitigen Zielplattformen in der gleichen Form abgebildet werden kann und dem Entwickler zur Verfügung steht.

Die Modellierung der Datenstruktur sollte sich möglichst nah an etablierten Sprachen, wie der UML oder ER Diagrammen orientieren. Hierdurch soll Entwicklern der Einstieg aufgrund ihres Vorwissens aus bekannten Domänen erleichtert werden.

3.1.2 Modellierung von Dateneigenschaften und -Behandlung

Neben der Spezifikation der Daten selbst gilt es ihre Eigenschaften und Behandlung in der späteren Datenschicht zu definieren. Je nach Anwendung und Domäne können für strukturell gleiche Datenmodelle vollständig unterschiedliche Eigenschaften gefordert sein.

Beispielsweise ändern sich in einer Anwendung Personendaten möglicherweise nur extrem selten und dürfen nur von einer zentralen Stelle bearbeitet werden. Hier könnte es ausreichend sein, sie einmal in der Woche zwischen Client und Server zu synchronisieren. In einer anderen Anwendung dagegen sind Personendaten ein zentrales Element und können stän-

dig von vielen verschiedenen Anwendern auch parallel offline bearbeitet werden. Hier sind eine möglichst häufige Synchronisation und eine Strategie zur Konfliktlösung bei widersprüchlichen Datenänderungen notwendig.

Um solche Anforderungen spezifizieren zu können muss eine domänenspezifische Sprache (DSL) entwickelt werden, die eine deutlich umfangreichere Ausdrucksmächtigkeit hat, als es die klassische Datenmodellierung, wie im letzten Abschnitt beschrieben, erlaubt. Mittels dieser DSL sollen Eigenschaften der Daten, wie zum Beispiel ihre Änderungsfrequenz, und daraus abgeleitet, Anforderungen an Ihre Behandlung, wie zum Beispiel Offlineverfügbarkeit, Synchronisationshäufigkeit oder Konfliktlösung definiert werden können. Für die Verwaltung dieser Zusatzinformationen zu den eigentlichen Nutzdaten sollen Meta-Daten zu dem Datenhaltungsmodell (bspw. Konsistenzbedingungen, nicht zu cachende Daten, Datenkontexte) vorgesehen werden.

Für umfangreiche Datenmodelle kann bereits die Modellierung der Daten selbst eine hohe Komplexität annehmen. Daher soll die DSL die Möglichkeit bieten unterschiedliche Sichten auf das Datenmodell zu erstellen. Durch die verschiedenen Sichten sollen die Entwickler nun jeweils andere Aspekte des Datenmodells bearbeiten und sich auf diese konzentrieren können. Beispielsweise ermöglicht eine Sicht die Modellierung der Datenstruktur selbst, eine andere stellt die Definition der Offline-Fähigkeit und der Updatefrequenz bereit, während eine dritte Sicht die Definition ermöglicht, wie mit Änderungen im Datenmodell beispielsweise bei einem Update auf eine neue Software-Version umgegangen werden soll. Welche Sichten letztendlich bereitstehen und von den Entwicklern genutzt werden, hängt von den jeweiligen Anforderungen ab. Innerhalb der Sichten soll dabei nach Möglichkeit auch eine Datenaggregation vorgenommen werden können. Liegen also beispielsweise Daten in einer Liste in einem Modell vor, so soll die Zahl der Listenelemente als aggregiertes Datum Teil der Sicht werden können.

3.1.3 Anforderungsübersicht

Die folgende Tabelle fasst die zuvor beschriebenen Anforderungen nochmals zusammen:

	Anforderung	Muss	Soll	Kann
A3.1-1	Datenstruktur-Modellierung angelehnt an etablierte Modellierungssprachen	X		
A3.1-2	Modellierung eines Datenmodells zentralen für alle Zielplattformen	X		
A3.1-3	DSL zur Spezifikation von Dateneigenschaften und –behandlung (ggf. Integration von Metadaten)	x		
A3.1-4	Möglichkeit getrennter Sichten zur Modellierung unterschiedlicher Aspekte	X		

3.2 Toolunterstützung

Im Rahmen von MOHITO wird eine für Softwareentwickler taugliche Toolchain umgesetzt, die neben einem Werkzeug zur möglichst komfortablen Instanziierung und Bearbeitung von MOHITO Modellen ["Data Layer Designer", A3.2-1] einen von Softwareentwicklern handhabbaren Generator zur Abbildung von MOHITO Modellen auf plattformspezifische Laufzeitartefakte umfasst [A3.2-2]. Um einen möglichst intuitiven Weg zur Bearbeitung der MOHITO Modelle zu gewährleisten, soll der Data Layer Designer dafür einen grafischen Editor anbieten [A3.2-3]. Endanwendertauglichkeit der Toolchain (relevant mit Hinblick auf Laufzeitänderungen des Datenmodells) und Reverse-Engineering werden im Projekt MOHITO nicht betrachtet. Die MOHITO Toolchain wird soweit wie möglich auf Eclipse aufbauen [A3.2-4].

	Anforderung	Muss	Soll	Kann
A3.2-1	Werkzeug zur Instanziierung und Bearbeitung von MOHITO Modellen	X		
A3.2-2	Generator zur Abbildung von MOHITO Modellen auf plattformspezifische Laufzeitartefakte	X		
A3.2-3	Nutzung der Eclipse-Plattform wo möglich	X		

3.3 Homogene Datenzugriffs-API

Ein wesentliches Ziel von MOHITO ist die Vereinheitlichung von Programmierschnittstellen über verschiedene Plattformen hinweg. Die Vereinheitlichung bezieht sich auf die technische Ebene (wenn möglich werden die gleichen Techniken über mehre Plattformen hinweg genutzt, bspw. REST), die fachliche Ebene der Schnittstellensignaturen (gleiche fachliche Dienste heißen gleich und nutzen die gleichen Parameter in der gleichen Reihenfolge), die Datentypen und Datenserialisierungen (bspw. JSON als Datenformat), die fachliche Protokollebene (gleiche Aufrufsequenzen, bspw. Initialisierung, Authentifizierung, Abruf, Terminieren auf allen Plattformen einheitlich) und die technische Zugriffsbene (bspw. REST, SOAP als Kommunikationsprotokoll). Das Ziel ist dabei eine Verringerung des plattformspezifischen Lernaufwands für Entwickler und ein konsistenter Funktionsumfangs über mehrere Plattformen hinweg.

Trotz des Ansatzes zur Homogenierung sollen dabei Plattformspezifika berücksichtigt werden können. Wenn eine serverseitige Datenhaltung bspw. keine Datenversionierung unterstützt, sollte dies durch einen Erweiterungspunkt im MOHITO Framework ergänzt werden können.

Datenoperationen für Datenobjekte sollen dabei mindestens CRUD Operationen (Create, Read, Update, Delete) umfassen. Darüber hinausgehende, anwendungsspezifische Services sollen spezifiziert und Code-Rümpfe hierfür generiert werden können.

Im Rahmen der Homogenisierung der Schnittstellen sollen dabei auch Qualitätseigenschaften der Schnittstellen (bspw. Dokumentation, sprechende Namen, Typsicherheit) berücksichtigt werden.

	Anforderung	Muss	Soll	Kann
A3.3-1	Vereinheitlichte Programmierschnittstelle über Plattformen hinweg	X		
A3.3-2	Möglichkeit zur Berücksichtigung von Plattformspezifika und Plattformeinschränkungen		X	
A3.3-3	Datenoperationen mit der Mächtigkeit von CRUD	X		
A3.3-4	Use Case spezifische Operationen, die über CRUD hinausgehen		X	
A3.3-5	Berücksichtigung der Schnittstellenqualität		X	

3.4 Laufzeitmechanismen

In diesem Abschnitt werden die technischen Anforderungen an die MOHITO Laufzeitumgebung dokumentiert. Die Anforderungen basieren auf einer Verallgemeinerung der Mechanismen, die zur Umsetzung der in Kapitel 2 beschriebenen Anwendungsszenarien notwendig sind. Die MOHITO Laufzeitmechanismen realisieren die Einhaltung der zum Entwurfszeitpunkt im MOHITO Datenmodell festgelegten Eigenschaften der Daten (vgl. Abschnitt 3.1).

3.4.1 Verteilung und Kommunikation

MOHITO setzt auf einer klassischen Client-Server Architektur für Geschäftsanwendungen auf. In dieser Architektur übernimmt ein Applikationsserver die zentrale Datenhaltung und komplexe Berechnungen über diesen Daten. Der Applikationsserver übernimmt außerdem die Datenversorgung für eine potentiell große Zahl heterogener und (teilweise) mobiler Clients über Protokolle wie SOAP oder das Architekturkonzept REST.

Die MOHITO Laufzeitumgebung ist eine Middleware-Lösung zum verteilten Zugriff auf die vom Applikationsserver bereitgestellten Daten. Sie bietet Mehrwertdienste für Caching [A3.4.1-1], Prefetching [A3.4.1-2] und Synchronisation von Datensätzen zwischen Client und Server [A3.4.1-3] an, auf deren Basis sich clientseitige Offline-Funktional umsetzen lässt [A3.4.1-4]. Die MOHITO Laufzeitumgebung muss mindestens REST unterstützen [A3.4.1-5]. Der Datenaustausch zwischen Client und Server erfolgt in MOHITO über strukturierte Datentypen in einer plattformunabhängigen Notation wie XML oder JSON [A3.4.1-6]. Mobile Clients greifen typischerweise über native Apps auf den Applikationsserver zu. Andere Clients nutzen den Browser für Zugriffe. Die MOHITO Laufzeitumgebung muss für mindestens eine

mobile Clientplattform (Android, iOS) verfügbar sein [A3.4.1-7/8] und sollte Browserunterstützung bieten [A3.4.1-9].

3.4.2 Basisfunktionalität des Applikationsservers

Der Applikationsserver bietet Clients CRUD (create, read, update, delete) Operationen an, mit denen einzelne Datensätze erzeugt, gelesen, modifiziert und gelöscht werden können. Neben einzelnen Datensätzen bietet der Applikationsserver auch eine Operation zum Lesen von Datensatzlisten an. Die MOHITO Laufzeitumgebung muss in der Lage sein, die geschilderten Zugriffsarten zu unterstützen [A3.4.2-1]. Der Aufbau der ausgetauschten Datensätze und ihre funktionalen und nicht-funktionalen Eigenschaften werden in dem MOHITO Datenmodell festgelegt. Der Applikationsserver implementiert typischerweise zusätzliche Funktionalität, um die Konsistenz, Vollständigkeit und referentielle Integrität der von ihm verwalteten Datensätze sicherzustellen. In diesem Zusammenhang können im Server Änderungen an den vom Client übermittelten Datensätzen vorgenommen werden. Die MOHITO Laufzeitumgebung muss in der Lage sein, mit solchen Änderungen umgehen zu können [A3.4.2-2]. Der Applikationsserver implementiert ein Berechtigungssystem, welches Zugriffsrechte für Datensätze verwaltet und durchsetzt. Die MOHITO Laufzeitumgebung muss in der Lage sein, sich mit dem vom Applikationsserver verwendeten Berechtigungssystem zu integrieren [A3.4.2-3].

3.4.3 Variabilität des Datenmodells

Das MOHITO Datenmodell ist über den Lebenszyklus einer Anwendung hinweg variabel. Unterschiedliche Versionen einer Anwendung können Unterschiede zwischen ihren Datenmodellen aufweisen, wobei in MOHITO von einem Ausbau des Datenmodells durch Hinzufügen neuer Datentypen und dem Erweitern existierender Datentypen um zusätzliche Attribute ausgegangen wird. Die MOHITO Laufzeitumgebung muss in der Lage sein, solche Änderungen zu unterstützen [A3.4.3-1]. Das Hinzufügen neuer Attribute zu einem existierenden Datentyp soll auch ohne einen Versionswechsel möglich sein. Dazu soll die MOHITO Laufzeitumgebung einen generischen Mechanismus bieten [A3.4.3-2].

3.4.4 Versionierung von Datensätzen

Änderungen eines existierenden Datensatzes führen zu unterschiedlichen Versionen eines Datensatzes. Bietet der Applikationsserver den Zugriff auf Datensatzversionen an, soll die MOHITO Laufzeitumgebung in der Lage sein, mit derart versionierten Datensätzen umzugehen [A3.4.4-1]). Andernfalls sollte sie selbst einen Erweiterungspunkt bereitstellen, über den Unterstützung für Datenversionierung ergänzt werden kann [A3.4.4-2].

3.4.5 Clientseitiges Caching

Die MOHITO Laufzeitumgebung muss einen clientseitigen Cache bereitstellen, der bereits vom Server an den Client ausgelieferte Datensätze vorhält [A3.4.5-1]. Lese- und Schreiban-

fragen des Clients werden immer dann aus dem Cache bedient, wenn der Server nicht erreichbar ist [A3.4.5-2]. Das Erzeugen und Löschen von Datensätzen im Cache soll ebenfalls möglich sein [A3.4.5-3]. Der Cache muss das automatische Verdrängen von Cacheeinträgen unterstützen, wobei die verwendete Verdrängungsstrategie (z.B. LRU, LFU) konfigurierbar sein soll [A3.4.5-4]. Einzelne Datensätze können durch eine entsprechende Markierung vor dem automatischen Verdrängen geschützt werden [A3.4.5-5]. Für diese Datensätze muss der Cache der Clientanwendung eine Schnittstelle anbieten, über die ein durch die Anwendung angestoßenes Verdrängen möglich ist [A3.4.5-5]. Ebenso muss es möglich sein, vom Server ausgelieferte Datensätze durch eine entsprechende Markierung vor clientseitigem Caching zu schützen [A3.4.5-6], um beispielsweise ihre Vertraulichkeit auch bei Verlust des Clients zu schützen oder ihre Aktualität zu garantieren. Zum Schutz vor unberechtigtem Zugriff soll die MOHITO Laufzeitumgebung eine Verschlüsselung der Cache-Inhalte unterstützen [A3.4.5-7].

Der Cache muss weiterhin eine Schnittstelle bieten, die einer Clientanwendung bzw. einem Systemadministrator Zugriff auf eine Cachestatistik und Meta-Informationen über Cache-Einträge (Alter des Eintrags, geändert, geschützt, …) anbietet [A3.4.5-8].

3.4.6 Datensynchronisation

Die MOHITO Laufzeitumgebung muss in der Lage sein, den Datenbestand im clientseitigen Cache mit dem des Applikationsservers durch einen Synchronisationsmechanismus abzugleichen [A3.4.6-1]. Während der Zwischenspeicherung eines Datensatzes im clientseitigen Cache können client- und serverseitig Änderungen des Datensatzes auftreten (Löschung, Modifikation von Attributwerten, Hinzufügen und Entfernen von Assoziationen mit anderen Datensätzen usw.), die zu Konflikten bei der Synchronisation führen können. Der Synchronisationsmechanismus muss in der Lage sein, konfligierende Änderungen zu erkennen [A3.4.6-2] und geeignet zu behandeln. Für die Konfliktbehandlung soll die MOHITO Laufzeitumgebung eine automatische und eine manuelle Konfliktlösungsstrategie unterstützen [A3.4.6-3]. Die MOHITO Laufzeitumgebung muss dafür u.a. Zugriff auf beide Versionen eines konfliktbehafteten Datensatzes bieten. Aufgrund der Einschränkungen von Smartphones und Tablets soll dort eine manuelle Konfliktbehebung nur für einzelne Datensätze angeboten werden [A3.4.6-4]. Eine manuelle Konfliktbehebung für große Mengen von Datensätzen soll dagegen auf einem PC und Notebook möglich sein [A3.4.6-5].

Der Zeitpunkt der Synchronisation richtet sich nach der Konnektivität des Clients. Die MOHITO Laufzeitumgebung muss deshalb den Status der Konnektivität des Clients erkennen können [A3.4.6-6]. Weiterhin sollte sie in der Lage sein, die QoS der Verbindung zu ermitteln, um den Synchronisationsvorgang darauf zu optimieren [A3.4.6-7]. Der Synchronisationsvorgang kann bei einer bestehenden Internetverbindung automatisch oder manuell angestoßen werden [A3.4.6-8]. Im ersten Fall wird mit der Synchronisation der im clientseitigen Cache befindlichen Datensätze begonnen, sobald eine Internetverbindung besteht. Im zwei-

ten Fall muss die MOHITO Laufzeitumgebung auch eine manuell angestoßene Synchronisation einzelner Datensätze unterstützen [A3.4.6-9].

Im mobilen Fall ist die Dauer der Internetverbindung typischerweise nicht vorhersehbar und die verfügbare Datenrate variabel. Um die Verbindung möglichst effizient zu nutzen, soll die MOHITO Laufzeitumgebung die Bestimmung der Reihenfolge, in der Datensätze synchronisiert werden, anhand von vorgegebenen Prioritäten vornehmen können [A3.4.6-10]. Sie muss ebenfalls mit einem plötzlichen Verbindungsabbruch während einer laufenden Synchronisation umgehen können [A3.4.6-11].

3.4.7 Prefetching und Offline-Management von Daten

Die MOHITO Laufzeitumgebung muss das Prefetching von Datensätzen in den clientseitigen Cache unterstützen (siehe Abschnitt 3.4.1). Welche Datensätze im Rahmen des Prefetching auf den Client geladen werden, wird von einer austauschbaren Prefetching-Strategie vorgegeben [A3.4.7-1]. Die MOHITO Laufzeitumgebung muss eine Prefetching-Strategie unterstützen, bei der die zu ladenden Datensätze manuell vom Nutzer ausgewählt werden. Weiterhin soll eine automatische Prefetching-Strategie unterstützt werden, die Datensätze über einen Algorithmus auswählt [A3.4.7-2]. Diese Auswahl kann beispielsweise über die Ermittlung der Relevanz von Datensätzen erfolgen, die den Kontext des Nutzers einbezieht. Weiterhin ist denkbar, dass ein solcher Algorithmus die von einem vorgegebenen Datensatz durch Verfolgen von Verknüpfungen erreichbaren Datensätze selektiert, wobei die Verknüpfungstiefe beschränkt ist.

3.4.8 Anforderungsübersicht

Die folgende Tabelle fast die zuvor beschriebenen Anforderungen an die MOHITO Laufzeitumgebung nochmals zusammen:

	Anforderung	Muss	Soll	Kann
A3.4.1-1	Unterstützung für Caching von Datensätzen	X		
A3.4.1-2	Unterstützung für Prefetching von Datensätzen	X		
A3.4.1-3	Unterstützung für Synchronisation von Datensätzen zwischen Client und Server	X		
A3.4.1-4	Offline-Funktionalität auf Client verfügbar	X		
A3.4.1-5	REST Unterstützung	X		
A3.4.1-6	Datenaustausch über strukturierte Datentypen in JSON oder XML Notation	X		
A3.4.1-7	Verfügbarkeit für eine mobile Plattform (Android oder iOS)	X		
A3.4.1-8	Verfügbarkeit für zwei mobile Platt-		X	

	formen (Android und iOS)			
A3.4.1-9	Browserunterstützung			X
A3.4.2-1	Unterstützung für CRUD-Zugriffe exisitiert	X		
A3.4.2-2	Serverinterne Modifikation übergebener Datensätze wird berücksichtigt	X		
A3.4.2-3	Integration mit dem Berechtigungssystem des Applikationsservers	X		
A3.4.3-1	Datenmodelländerungen über den Lebenszyklus einer Anwendung hinweg unterstützen	X		
A3.4.3-2	Hinzufügen neuer Attribute zu existierendem Datentyp zur Laufzeit möglich	X		
A3.4.4-1	Versionierte Datensätzen werden unterstützt		X	
A3.4.4-2	Erweiterungspunkt für Datensatzversionierung durch MOHITO Runtime existiert			X
A3.4.5-1	Clientseitiger Datensatzcache	X		
A3.4.5-2	Cache bedient Lese-/Schreibanfragen falls Server nicht erreichbar	X		
A3.4.5-3	Erzeugen und Löschen von Datensätzen im Cache möglich		X	
A3.4.5-4	Automatisches Verdrängen von Cacheeinträgen (konfigurierbar)	X		
A3.4.5-5	Datensätze lassen sich durch Markierung vor automatischer Verdrängung schützen (Verdrängung erfolgt durch expliziten Aufruf)	X		
A3.4.5-6	Datensätze lassen sich durch Markierung vor clientseitigem Caching schützen	X		
A3.4.5-7	Cache-Inhalte werden verschlüsselt		X	
A3.4.5-8	Cache liefert Cache-Statistik und Metadaten zu Cacheeinträgen über eine Schnittstelle aus	X		
A3.4.6-1	Synchronisation zum Datenabgleich zw. Cache und Server möglich	X		
A3.4.6-2	Synchronisationskonflikte werden erkannt	X		
A3.4.6-3	Sync.-konfliktbehandlung ist automatisch und manuell möglich	X		
A3.4.6-4	Manuelle Konfliktbehebung auf mobilen Clients für einzelne Datensätze	X		

A3.4.6-5	Manuelle Konfliktbehandlung für mehrere Datensätze über Desktop/Notebook möglich		X	
A3.4.6-6	Konnektivität des Clients wird erkannt	X		
A3.4.6-7	QoS der Internetverbindung wird bei Synchronisation berücksichtigt			X
A3.4.6-8	Start des Sync.-vorgangs bei bestehender Verbindung wahlweise automatisch oder manuell	X		
A3.4.6-9	Manuelle Synchronisation einzelner Datensätze bei manuellem Start des Sync.-vorgangs möglich	X		
A3.4.6-10	Synchronisationsreihenfolge von Datensätzen über Prioritäten steuerbar		X	
A3.4.6-11	Verbindungsabbruch bei laufender Synchronisation wird geeignet behandelt	X		
A3.4.7-1	Datensatzauswahl für Prefetching über austauschbare Strategie möglich	X		
A3.4.7-2	Prefetching-Strategie mit manueller Datensatzauswahl existiert	X		
A3.4.7-3	Prefetching-Strategie mit automatischer Datensatzauswahl existiert		X	

3.5 Anbindung von Bestandstechnik

Für die Vermittlung von Daten von und zu angebundenen mobilen Clients soll, eine entsprechende Eignung bei der Detailanalyse vorausgesetzt, das aus dem Theseus-Projekt stammende MML (Mobility Mediation Layer) an die MOHITO-Plattform angekoppelt werden.

Das ähnlich zu MOHITO gelagerte Projekt Modagile Mobile bietet einen Datenhaltungs-Stack für mobile Anwendungen, beginnend in der Benutzeroberfläche. Die Serveranbindung ist hier nur begrenzt ausgeprägt. Um eine Zusammenarbeit zwischen MOHITO und Modagile Mobile zu ermöglichen soll der Modagile Mobile Datenhaltungs-Stack sowie das Modagile Mobile zu Grunde liegende Datenmodell angebunden werden.

Weiterhin soll eine Anbindung von in MOHITO entwickelten Lösungen an die Plattform CAS Open und die darauf aufsetzende SaaS-Lösung CAS PIA, einschließlich der dafür verfügbaren mobilen Apps, erfolgen.

	Anforderung	**Muss**	**Soll**	**Kann**
A3.5-1	Anbindung an das MML		X	
A3.5-2	Anbindung an die Modagile Mobile Datenhaltung und Integration in die Werkzeugkette		X	

A3.5-3	Anbindung an CAS Open		X	

3.6 Nicht-technische Anforderungen

Die MOHITO Projektergebnis sollen losgelöst von der im Projekt verwendeten Bestandstechnik verwertbar sein. Dazu soll ein Referenzstack ohne proprietäre Anteile als Open Source Lösung verfügbar gemacht werden [A3.6-1]. Damit ist ihre Weiternutzung nach Projektende ohne Produktkauf möglich.

	Anforderung	**Muss**	**Soll**	**Kann**
A3.6-1	Bereitstellung eines Referenzstacks als Open Source		X	

4 Stand der Technik im Umfeld der Anforderungen

4.1 Plattformübergreifende Datenmodellierung

Datenmodellierung, wie sie heutzutage praktiziert wird, kann aufgrund ihrer Implementierungsnähe und der Entwurfsunterstützung in drei verschiedene Abstraktionsstufen, aufgrund ihrer Implementierungsnähe und der Entwurfsunterstützung, gegliedert werden. Wie in Abbildung 2 dargestellt reicht dies von der direkten Implementierung, über die einfache Beschreibung der Daten bis hin zur Modellierung der Daten.

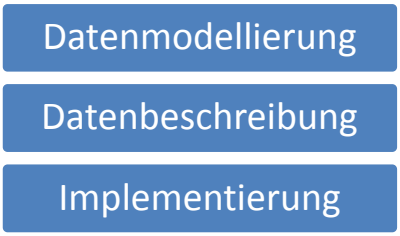

Abbildung 2- Abstraktionsebenen der Datenmodellierung

4.1.1 Implementierung

Auf der untersten Ebene, der Implementierung wird das Datenmodell direkt im Code ausprogrammiert. Je nach Programmiersprache stehen hierbei Konzepte wie Interfaces zur Verfügung, die eine Trennung der Datenstruktur (Entitäten, Attribute, Referenzen) und der eigentlichen Implementierung und deren Charakteristika (Serialisierbarkeit, Hintergrundverarbeitung, etc.) erlauben. Beispiele hierfür sind die Klassischen Interfaces der objektorientierten Programmiersprachen, wie Java oder C#.

4.1.2 Datenbeschreibung

Datenbeschreibung ist keine direkte Programmierung im klassischen Sinne, sondern beschreibt ein Datenschema, aus dem die Implementierung erzeugt werden kann. In diese Klasse der Datenmodellierung sind vor allem Beschreibungen mit textueller Notation einzuordnen.

Hierzu gehören zum einen die klassische Data Definition Language (DDL) des Structured Query Language (SQL) Standards. Einem Standard speziell aus dem Datenbankumfeld, der für die Erzeugung von Datenbank Schemata inklusive der Tabellen, Triggern und ähnliches dient. Neben dem plattformübergreifenden Standard, der von den meisten SQL Servern in unterschiedlichem Umfang unterstützt wird, verfügt nahezu jeder Hersteller eines SQL Server Produktes über eine plattformspezifische Anpassung. Zu den bekanntesten kommerziellen Produkten gehören Microsoft SQL Server, Oracle und DB2. Im Open Source Bereich sind dies vor allem MySQL, PostgeSQL und SQL Lite. Während die Definition des Datenschemas selbst sich rein auf die Datenstruktur bezieht, können beispielsweise durch Constraints, Indizes, Trigger und Prozeduren auch Verarbeitungsmechanismen definiert werden.

Neben den Datenbank-spezifischen Beschreibungen existieren verschiedene Standards und Beschreibungssprachen, wie der OMG Standard CORBA Interface Definition Language (CORBA-IDL) und der W3C Standard Webservice Description Language (WSDL), die eine Modellierung eines Datenmodels im Rahmen einer Schnittstellendefinition anbieten. Diese Standards sind in der Regel Tool- und Plattform-unabhängig. Viele Plattformen bieten jedoch Generatoren, um aus einer entsprechenden Beschreibung die Implementierung eines Datenmodells zu generieren. Beschreibungssprachen, wie die WSDL und der CORBA-IDL Standard bieten nur die Definition der Schnittstelle, bzw. des Datenmodells selbst. Sie machen in der Regel keine Angaben über die dahinterliegende Implementierung auf einer Plattform.

Einen Mix aus diesen beiden Typen stellen Datenbeschreibungssprachen sogenannter Object-Relational-Mapper (OR-Mapper) dar. OR-Mapper dienen der Übersetzung zwischen relationalen Datenbanken und objektorientierter Anwendungssoftware. Da OR-Mapper neben grundlegenden Operationen wie dem Lesen und Schreiben von Daten auch Funktionalität wie Caching übernehmen können, bieten ihre individuellen Beschreibungssprachen auch die Möglichkeit diese zu definieren. Die meisten OR-Mapper bieten neben der Generierung des Datenbankschemas als allgemeine SQL Definition, sowie der Generierung der Implementierung als Programmcode, auch die Anbindung unterschiedlicher SQL Server sowie deren spezifischen Datenformate und technischen Schnittstellen. Wichtig an dieser Stelle ist jedoch, dass sich OR-Mapper auf die Verbindung zwischen objektorientierter Software und relationalen Datenbanken fokussieren. Eine generelle, plattformübergreifende Datenmodellierung inklusive der Kommunikation und technischen Unterstützung auf unterschiedlichen Plattformen, die über SQL Server hinausgehen, ist hierbei nicht vorgesehen. Bekannte Vertreter von OR-Mappern, speziell im Java Umfeld sind Hibernate und die darin enthaltene Hibernate Query Language (HQL) (www.hibernate.org) sowie MyIBatis (www.mybatis.org).

4.1.3 Datenmodellierung

Unter Datenmodellierung ist eine leistungsfähigere Datenbeschreibung zu verstehen. Hierbei werden visuelle Darstellungen zur einfacheren Modellierung im Vergleich zu der in 4.1.2 beschriebenen textuellen Datenbeschreibung verwendet. Darüber hinaus wird durch eine höhere Abstraktion, weg von Details der Umsetzung, genutzt, um die Komplexität zu reduzieren und den Fokus auf das eigentliche Datenmodell zu erhöhen.

Datenmodellierung mittels grafischen Editoren und domänenspezifischen Visualisierungen ist bereits seit langem eine etablierte Praxis in der Software Entwicklung. Während in der Vergangenheit jedoch eine starke Bindung an die Zielplattform bestand – beispielsweise Chen Diagramme speziell für das Datenbankumfeld - so liegt der Fokus inzwischen mehr auf der Modellierung von Datenmodellen angelehnt an die Problemdomäne anstatt einer technischen (Teil-)Domäne. Dies hat auch dazu geführt, dass Standardsprachen, wie die Unified Modeling Language (UML) inklusive verschiedener anforderungsspezifischer Profile, oder dem Eclipse Modeling Framework und dem Ecore Standard entstanden sind. Sie erlauben eine Modellierung von Datenstrukturen unabhängig von der späteren Realisierungstechnik. Eine Unterstützung dieser Sprachen bieten heutzutage Open-Source Werkzeuge, wie das Eclipse Modeling Projekt oder kommerzielle Produkte, wie der Rational Software Architect (RSA) von IBM oder der Enterprise Architect von Sparx Systems.

4.1.4 Transformation zwischen den Abstraktionsschichten

Für alle oben beschriebenen Abstraktionsschichten gilt, dass die jeweils darunter befindlichen Schichten daraus generiert werden können. Wie teilweise bereits im Text beschrieben ist, ist es beispielsweise möglich aus einer Datenbeschreibung eine Implementierung des Datenmodells, sei es für eine Programmiersprache, oder in einem SQL Server zu erzeugen. Ebenso können aus Datenmodellen, wie sie in Abschnitt 4.1.3 beschrieben sind, sowohl Datenbeschreibungen als auch direkt Implementierungen generiert werden. Bei beiden Arten der Datengenerierung stecken die Regeln, wie die notwendigen Details zu ergänzen sind, in der jeweiligen Transformation, die oftmals auch als Generator bezeichnet wird.

Liegt bereits ein Datenmodell auf einer niedrigeren Abstraktionsebene vor, so unterstütze viele Werkzeuge auch das als Reverse-Engineering bezeichnete erzeugen eines Modells auf einer höheren Abstraktionsebene. Sowohl kommerzielle Modellierungsumgebungen, wie zum Beispiel die DB Suite von Embarcadero als auch kostenlose Werkzeuge wie die MySQL Workbench erlauben ein solches Reverse-Engineering. Zu beachten ist hierbei jedoch, dass ein solches Reverse-Engineering in der Regel nur für eingeschränkte Plattformen möglich ist. So bieten die beiden genannten Werkzeuge auch nur einen Import von Datenbankbeschreibungen, sei es als SQL oder direkt aus dem SQL Server in ein ER Modell. Ebenso existieren Werkzeuge, wie MoDisco aus dem Eclipse Modeling Projekt, oder der IBM Rational Architect, die aus einer bestehenden Java Implementierung ein UML Modell erzeugen können.

Die genannten Werkzeuge sind nur Beispiele für eine sehr große Bandbreite von existierenden Lösungen. Da es jeweils Plattform-, Beschreibungssprachen- und Modell-spezifischer Transformationsregeln bedarf, ist ein generisches Werkzeug, das beliebige Transformationen bereitstellt, auch nicht möglich. Nichts destotrotz sind solche Transformationen sowohl für eine Neu-Entwicklung mittels modelgetriebener Technologien als auch eine Migration bestehender Software hin zu einem modellgetriebenen Ansatz nur durch solche Transformationen möglich.

Ein Sonderfall bei der Verarbeitung von Datenmodellen und Beschreibung innerhalb einer Software ist die Interpretation des Datenmodells bzw. der Datenbeschreibung. Aber auch hier gilt, dass Regeln existieren müssen, wie das Datenmodell- bzw. die Beschreibung in der Implementierung zu verarbeiten ist. Diese Regeln befinden sich bei der Interpretation jedoch nicht in einem Generator, sondern in der Implementierung selbst und werden zur Laufzeit und nicht zur Entwicklungszeit ausgewertet. Für die generellen Konzepte der Datenmodellierung hat dies jedoch keine Auswirkung.

4.2 Laufzeitmechanismen

Nachfolgend soll auf schon vorhandene Technologien und Frameworks im Bereich der Datenpersistenz sowohl für mobile Plattformen als auch für den Serverbereich eingegangen und existierende Lösungen auf ihre Relevanz für das MOHITO Projekt evaluiert werden. Des Weiteren wird der Problembereich der Datensynchronisation und entsprechender Konfliktlösungsstrategien betrachtet. Abschließend werden wir uns dem Thema Caching widmen, um auch hier den Stand der Technik zu erfassen.

4.2.1 Persistenzschichten für Serverumgebungen

Im Java Umfeld, in dem wir uns im MOHITO Projekt bewegen, hat sich im Laufe der Zeit eine Vielzahl unterschiedlicher Techniken und Frameworks zum Thema Datenpersistenz entwickelt. Dies ist zum einen der großen Beliebtheit und Verbreitung der Sprache zuzuschreiben, zum anderen aber auch der immer wiederkehrenden Notwendigkeit geschuldet, Datenobjekte zu persistieren.

Wenn über Persistenz im Java Umfeld diskutiert wird, ist unbedingt ein Entwurfsmuster anzuführen, was ursprünglich von Sun erdacht und in die „Core J2EE Patterns“ [3] aufgenommen wurde. Es handelt sich um das „Data Access Object“ (DAO) Entwurfsmuster. Unter einem Daten Zugriffs Objekt (DAO) versteht man ein Objekt, das eine abstrakte Schnittstelle zu einem dahinterliegenden Persistenzsystem bietet. Solch ein Persistenzsystem kann eine Datenbank oder einfach eine Datei sein, in welcher die Zustandsdaten eines Objekts binär oder in einem anderen Format, beispielsweise XML, gespeichert werden. Durch das DAO Interface ist das zugrunde liegende Persistenzsystem völlig von der Anwendungslogik ge-

trennt und kann jederzeit ausgetauscht werden. Es wurde somit in der Persistenzschicht eine Unabhängigkeit der Anwendungslogik von der konkreten Datenhaltung erreicht.

In der Java Welt kann nun das DAO Entwurfsmuster in einer Vielzahl von Formen implementiert werden. Dies reicht von einfachen Schnittstellen, die den Datenzugriff von der Anwendungslogik trennen, bis hin zu freien und kommerziellen Frameworks, welche eine Anbindung an die unterschiedlichsten Persistenzsysteme bieten. So stellt DAO zwar nur ein abstraktes Entwurfsmuster dar, das allerdings in Java-spezifischen Technologien wie der „Java Persistence API“ oder der „Java Data Objects“ (JDO) umgesetzt wurde und durch diese Technologien recht einfach einsetzbar ist.

4.2.1.1 Java Data Objects

Die Java Data Objects (JDO) API stellt eine Spezifikation zur persistenten Speicherung von simplen Java Objekten dar. Sie wurde durch die „Object Database Management Group“ (ODMG) beeinflusst und von einem Konsortium bestehend aus Sun, IBM und Apple verabschiedet. Für die Version 1 hat Sun sowohl die Spezifikation als auch eine Referenzimplementierung bereitgestellt. Seit Version 2 wird der Standard von der „Apache Software Foundation“ im Rahmen des „Apache JDO open-source project“ betreut. [4] Die Spezifikation der JDO stellt ein Framework mit einer einheitlichen Schnittstelle zum Managen persistenter Daten zur Verfügung. Hierbei ist die zugrundeliegende Speichertechnologie völlig transparent und austauschbar. Speichermedien können Datenbanksysteme oder auch ein Dateisystem sein. Die Objektpersistenz wird in Form von Metadaten in XML Dateien definiert.

Die wichtigste Implementierung des JDO Standards ist die „DataNucleus AccessPlatform“, welche eine gewisse Bedeutung erlangt hat, da sie als Persistenzschicht für die Google App Engine fungiert.

4.2.1.1.1 DataNucleus

DataNucleus ist ein Open Source Projekt, das eine Fülle von Softwareprodukten zum Management von Anwendungsdaten unter Java zur Verfügung stellt. Das für diese Betrachtung relevante Produkt ist die „DataNucleus AccessPlatform“, welche größtmögliche Flexibilität in Bezug auf unterstützte Persistenzstandards und Datenspeicher bietet. Die „DataNucleus AccessPlatform“ stellt eine vollständig konforme Implementierung der Java Data Objects (JDO) 1.0, 2.0, 2.1, 2.2, 3.0, 3.1 Spezifikationen und der Java Persistence API 1.0, 2.0 Spezifikationen dar und deckt somit die größtmögliche Bandbreite an Persistenzstandards im Java Bereich ab. Als Datenspeicher werden alle gängigen relationalen Datenbanksysteme, etliche objektorientierte Datenbanken und andere Datenspeicher wie Excel (XLS/OOXML) und LDAP unterstützt. [5] Dies macht DataNucleus zu einem mächtigen und vor allem flexiblen Framework, was auch zu einer weiten Verbreitung geführt hat.

4.2.1.2 Java Persistence API

Die Java Persistence API (JPA) ist ebenfalls ein Java Persistenz Standard, der aber im Gegensatz zu JDO Speichertechnologie agnostisch ist, und nur das Persistieren von simplen Java Objekten in objektrelationale Datenbanken vorsieht. Der Standard wurde von der EJB 3.0 Expert Group entwickelt und wurde im Mai 2006 erstmals veröffentlicht. [6] Er kann heute im Java Umfeld als wichtigster Persistenz Standard angesehen werden, für den es etliche Implementierungen gibt, auf die später noch genauer eingegangen wird.

Der Standard definiert Persistenz über ein objektrelationales Mapping. Das bedeutet Java Objekte werden durch definierte Regeln auf relationale Datenbanken abgebildet und können von dort auch wieder geladen und instanziiert werden. Objekte werden in der JPA durch sogenannte Entitäten repräsentiert. Eine Entität entspricht typischerweise einer Java Klasse. Solch eine Entität wird durch eine Tabelle in einer zugrundeliegenden relationalen Datenbank dargestellt. Eine Instanz einer Entität, also ein simples Java Objekt, ist dann eine Zeile in dieser Tabelle. Die Beschreibung, welche Felder einer Klasse wie und auf welche Spalten abgebildet werden, kann in Form von XML Metadaten oder per Java Annotation realisiert werden.

Im Folgenden soll auf die zwei wichtigsten Implementierungen der JPA eingegangen werden. Dies ist zum einen EclipseLink, die Referenzimplementierung für die Java Persistence API 2.0 und zum anderen Hibernate ein sehr weit verbreitetes, quelloffenes Framework.

4.2.1.2.1 EclipseLink

EclipseLink ist ein quelloffenes Persistenz Projekt der „Eclipse Foundation". Es bietet ein erweiterbares Framework, das es Java Entwicklern erlaubt mit unterschiedlichsten Datendiensten, wie Datenbanken, XML Repräsentationen oder Enterprise Informations-Systemen (EIS) zu interagieren. EclipseLink unterstützt eine Reihe von Persistenzstandards, unter anderem die Java Persistence API (JPA) und die Java API für XML Binding (JAXB) und stellt die Referenzimplementierung für die Java Persistence API 2.0 dar. [7] Es basiert auf dem kommerziellen Produkt „TopLink", dessen Quellcode Oracle größtenteils für den Start des EclipseLink Projekts beigesteuert hat. Das Projekt stellt eine gewisse strategische Bedeutung für Oracle dar, da die nächsten Versionen des „Oracle Application Servers" sowie des „GlassFish 3" open-source Anwendungsserver, der von Oracle gesponsert wird, auf EclipseLink basieren sollen. [8]

4.2.1.2.2 Hibernate

Das Community-getriebene Projekt Hibernate bietet ein quelloffenes Persistenz und ORM-Mapper Framework, welches das Abbilden von regulären Java Objekten auf relationale Datenbanken ermöglicht. Damit können Objekte mit Attributen und Methoden (im Java-Bereich

POJOs genannt) in relationalen Datenbanken gespeichert werden. Aus diesen Datensätzen können später wieder auf umgekehrtem Weg Objekte erzeugt werden. Beziehungen zwischen Objekten werden durch entsprechende Datenbankrelationen abgebildet. [9] Hibernate bietet eine Anbindung an alle gängigen relationalen Datenbanken bzw. deren SQL Dialekte an. Unter anderem werden Oracle, DB2, MS SQL Server, Sybase, PostgreSQL, MySQL und SAP DB unterstützt [10]. Es zeichnet sich darüber hinaus durch seine einfache Benutzbarkeit und steile Lernkurve aus, da es weniger komplex als vergleichbare Frameworks ist, was unter anderem auch die weite Verbreitung von Hibernate erklärt.

4.2.2 Persistenzschichten für mobile Plattformen

Im Bereich der mobilen Endgeräte haben sich grundsätzlich zwei Ansätze der Datenpersistenz durchgesetzt. Zum einen ist dies das Speichern von Daten über Key-Value-Stores, zum anderen die Verwendung an mobile Anforderungen angepasster Datenbanken wie SQLite unter Android und iPhone oder SQL Server CE unter Windows Phone 7. Im Folgenden werden beide Ansätze beschrieben und ihre Verwendung auf den einzelnen Plattformen näher beleuchtet.

4.2.2.1 Key-Value-Stores

Unter den drei mobilen Plattformen Android, iPhone und Windows Phone, die in diesem Zusammenhang genauer betrachtet werden sollen, haben sich die sogenannte Key-Value-Stores zum Speichern simpler, unstrukturierter Daten etabliert. Die API's zum Managen der Daten und ihrer physikalische Speicherung auf den jeweiligen Plattform unterscheiden sich zwar, das zugrundliegende Konzept ist aber das gleiche. Daten, die kein relationales Modell der Datenstrukturierung benötigen, also keine Abhängigkeiten untereinander modellieren und auch keine dynamischen Anfragen unterstützen müssen, lassen sich einfacher und performanter in Key-Value-Stores Key-Value-Stores, zu denen auch die sogenannten NoSQL Datenbanken gehören, ablegen.

Daten, die sich gut durch Key-Value Paare darstellen lassen, sind im Kontext mobiler Anwendungen meist Einstellungen oder Nutzerpräferenzen. Entsprechend sind die mobilen Schnittstellen zum Speichern und Abfragen dieser Key-Value Paare auch in der Namensgebung im Bereich der Nutzereinstellungen angesiedelt und heißen „Shared Preferences" (Android), „NSUserDefaults" (iPhone) oder IsolatedStorageSettings (Windows Phone).

Windows Phone 7 bietet das IsolatedStorage Konzept an, um anwendungsspezifische Daten sicher zu speichern. Die Sicherheit wird dadurch erlangt, dass Daten einer Anwendung nur von dieser selbst gelesen und modifiziert werden dürfen. Anwendungen haben keinen Zugriff auf Daten, welche nicht zu Ihrer Domäne gehören. Die Daten der einzelnen Anwendungen sind somit voneinander isoliert. Für das Speichern von Key-Value Paaren kann das Isola-

tedStorageSettings Interface genutzt werden, welches den IsolatedStorage zum Speichern von Key-Value Paaren nutzt. [11]

Unter iPhone kann das „user defaults system“ genutzt werden. „User defaults“ stellen auch hier eine Anwendungsspezifische Domain dar, auf die nur die Anwendung selbst Zugriff hat. Verwaltung und Zugriff auf die Daten wird durch das „NSUserDefaults“ Interface gewährleistet. Technisch sind die Daten in XML basierten Dateien gespeichert, die nach der Anwendungsdomäne benannt sind. [12]

Bei Android gibt es das „Shared Preferences“ Framework, welches ebenfalls das Speichern von Key-Value Paaren erlaubt. Daten können über eine „Shared Preferences Editor“ Klasse manipuliert und gespeichert werden. Auch bei diesem Ansatz sind die Daten in XML basierten Dateien persistiert, die in einem geschützten Bereich der Anwendung liegen und nur von dieser gelesen und geschrieben werden dürfen. [13]

4.2.2.2 Datenbanksysteme auf mobilen Endgeräten

Aufgrund der eingeschränkten Ressourcen mobiler Plattformen in Bezug auf Rechenleistung, Arbeitsspeicher, Kapazität des nichtflüchtigen Datenspeichers und Stromverbrauch, müssen Datenbanken, die auf solchen Plattformen zum Einsatz kommen, an diese Gegebenheiten angepasst sein. Meist sind mobile, relationale Datenbankmanagementsysteme in ihrem Speicherbedarf aber auch im Funktionsumfang kleiner als Ihre Gegenstücke im Desktop und Serverbereich. So sind Mobile Datenbanksysteme beispielsweise oft Einzelbenutzersysteme, die daher keine Steuerung für gleichzeitige Datenzugriffe benötigen. Auch Wiederherstellungsmechanismen oder die Verwaltung von Benutzer und Zugriffsberechtigungen fehlen häufig oder sind gewissen Einschränkungen unterworfen.

Anfänglich haben sich auf den verschiedenen Mobilen Plattformen unterschiedliche Datenbanksysteme verbreitet. So hat beispielsweise Sybase mit seinem Produkt „SQL Anywhere 11“ den Blackberry Markt bedient, wohingegen Microsoft für seine mobilen Betriebssysteme Windows CE und Windows Phone die SQL Server Compact Edition entwickelt hat. Doch mit dem Aufkommen des kleinen leistungsfähigen Datenbanksystems „SQLite“ hat sich in den letzten Jahren ein de facto Standard auf allen mobilen Systemen durchgesetzt. Spätestens, nachdem bekannt wurde, dass SQLite auch für Windows Phone 8 verfügbar sein wird [14], kann man nun behaupten, dass SQLite die wichtigste und meistgenutzte Datenbank auf mobilen Endgeräten darstellt.

4.2.2.2.1 SQLite

SQLite ist eine kleine, schnelle und zuverlässige Datenbank. Sie wurde ursprünglich von D. Richard Hipp im Frühjahr 2000 entwickelt und untersteht der Public Domain. SQLite wird seither ständig weiterentwickelt. Ein Konsortium aus Firmen, zu denen unter anderem

Oracle, Adobe und Mozilla gehören, unterstützt die Weiterentwicklung und setzt gleichzeitig SQLite in ihren eigenen Produkten ein. [15]

Zum Erfolg und der Verbreitung von SQLite, gerade auf mobilen Plattformen, haben sicher zwei wichtige Faktoren beigetragen. Zum einen der äußerst kleine Speicherbedarf von nur ca. 350 KB [16], den das Datenbankmanagementsystem von SQLite benötigt. Zum anderen ein plattformunabhängiges Dateiformat, in dem die Daten persistiert werden. Die gesamte Datenbank ist in einer gewöhnlichen Datei im Dateisystem des zugrundeliegenden Betriebssystems abgelegt. Diese Datei kann zwischen Betriebssystemen mit unterschiedlichen Architekturen wie Big-endian oder Little-endian, bzw. 32-bit oder 64-bit, ausgetauscht werden ohne dabei die Datei ändern zu müssen. Darüber hinaus ist das Dateiformat abwärtskompatibel, so dass auch neuere Versionen von SQLite noch mit älteren Datenbankdateien umgehen können.

SQLite ist von seinem gesamten Grundkonzept sehr einfach gehalten, kompakt im Speicherverbrauch und plattformübergreifend ausgelegt. Dies unterstreicht auch die Tatsache, dass die Datenbank ohne jegliche Konfiguration auskommt. Es gibt nichts, was vor der Benutzung installiert oder konfiguriert werden müsste. Außerdem setzt die Datenbank auch nicht auf einer Client/Server-Architektur auf, wie es bei Datenbanksystemen sonst üblich ist. Es fällt daher sämtlicher Zusatzaufwand weg, welcher sonst durch die Kommunikation mit einem dedizierten Serverprozess anfallen würde, was SQLite noch einmal stark vereinfacht und wiederum für den Einsatz auf mobilen Systemen prädestiniert.

Abschließend soll noch eine Auswahl an mobilen Plattformen angeführt werden, auf welchen SQLite verfügbar ist: [17]

- Windows Phone 8 (Microsoft)
- Android (Google)
- iOS (Apple)
- Symbian OS (Symbian)
- Maemo (Nokia's)
- BlackBerry (RIM)

4.2.2.3 Persistenzframeworks auf mobilen Plattformen

Im Bereich mobiler Plattformen haben sich leider noch keine einheitlichen Persistenzframeworks durchgesetzt. Diese werden auch von einem Großteil der oft recht simplen Anwendungen gar nicht benötigt. Oft genügt es hier schon die plattformeigenen Key-Value-Stores zum einfachen Persistieren von Nutzer- oder Programmdaten zu verwenden. Vereinzelt allerdings müssen größere und komplexere Datenmengen verwaltet werden. In diesem Fall kommen dann meist SQLite Datenbanken zum Einsatz, die je nach Plattform, entweder direkt angesprochen oder über systemeigene Schnittstellen verwendet werden. Diese Schnitt-

stellen erleichtern allerdings nur die Datenbanknutzung, indem sie beispielsweise Datenbankzugriffe durch Wrapper-Funktionen vereinfachen. Sie können aber nicht mit dem Funktionsumfang objektrelationaler Persistenzframeworks im Serverbereich verglichen werden.

Hier kommt natürlich die Frage auf, warum auf Plattformen, die Java unterstützen oder im Fall von Android, wo Java gar die bevorzugte Entwicklungssprache ist, nicht die für Java vorhandenen Persistenzframeworks genutzt werden können. Die Erklärung hierfür ist, dass sich die zuvor vorgestellten JPA und JDO Frameworks nicht so einfach auf die entsprechenden Plattformen portieren lassen. Der Grund liegt in der Arbeitsweise der Persistenzframeworks. Sie erweitern oder manipulieren nach dem Kompilieren oder zur Laufzeit den Java Bytecode, um die Persistenzmechanismen zu realisieren. Diese Frameworks arbeiten somit alle auf dem Bytecode, welcher von der im Desktop- und Serverbereich eingesetzten „Java Virtual Machine“ (JVM) ausgeführt wird. Auf mobilen Plattformen kommt jedoch nicht die Java Standard Edition mit der JVM als Laufzeitumgebung zum Einsatz, sondern andere, spezialisierte Java Versionen bzw. Laufzeitumgebungen. Bei vielen Herstellern wird hier die „Java Platform Micro Edition“ (Java ME) genutzt. Sie bringt als Laufzeitumgebung die KVM mit [18], welche nicht mit den Bytecode Erweiterungen der Persistenzframeworks umgehen kann. Im Fall von Android wird eine von Google eigens entwickelte Laufzeitumgebung namens Dalvik genutzt. Die Dalvik VM arbeitet mit .dex (Dalvik Executable) Dateien, die entsprechenden Dalvik Bytecode enthalten, der ebenfalls inkompatibel zum Bytecode der Java Virtual Machine ist.

Wie schon erwähnt gibt es sowohl für Android als auch für iOS Frameworks, die grundlegende Funktionalitäten zur Datenpersistenz bereitstellen, auch wenn sie nicht den gleichen Komfort der Persistenzframeworks im Serverbereich bieten können. Zwei dieser Frameworks, eins für Android und eines für iOS, sollen im Folgenden näher betrachtet werden.

4.2.2.3.1 ActiveAndroid

Das ActiveAndroid Framework bietet schon sehr viel von dem, was man von einem Persistenzframework im Serverbereich erwartet. Es realisiert objektrelationales Mapping durch Quellcode Annotationen, so wie man es auch von der „Java Persistence API“ her kennt. Auch das Abbilden von Relationen einzelner Klassen untereinander ist vorgesehen. So können Eins-Zu-Eins oder Eins-Zu-Viele Beziehungen durch entsprechende Annotationen dargestellt werden. Somit stehen die rudimentären Funktionalitäten eines Persistenzframeworks zur Verfügung.

Im Vergleich zu seinen Pendants im Serverbereich zeichnet sich ActiveAndroid durch seine Einfachheit aus, die allerdings auch eine eingeschränkte Funktionalität mit sich bringt. ActiveAndroid ist keine Implementierung eines offenen Persistenzstandards wie der JPA oder JDO, welcher jederzeit austauschbar oder erweiterbar wäre, sondern stellt eine proprietäre

Entwicklung dar, die auf dem Active Record Entwurfsmuster basiert. Daher leitet sich auch der Name des Frameworks ab. Das Entwurfsmuster wurde von Martin Fowler in seinem Buch „Patterns of Enterprise Application Architecture“ [19] beschrieben und stellt eine Möglichkeit objektorientierter Software dar Daten in einer relationalen Datenbank zu speichern. Das Muster sieht vor, dass Klassen, die persistiert werden sollen, Schnittstellen zum Einfügen, Ändern und Löschen implementieren, die sich dann auf eine Datenbanktabelle beziehen. Ein Objekt, bzw. eine Instanz dieser Klasse entspricht dann einer Zeile in der Tabelle. Das ActiveAndroid Framework stellt die Implementierung dieses Entwurfsmusters in Basisklassen bereit, von denen die zu persistierenden Klassen ableiten können, um so recht einfach ein objektrelationales Mapping zu realisieren.

4.2.2.3.2 iOS Core Data

Core Data ist ein Objekt-Graph Management- und Persistenzframework. Es stellt Mechanismen zur Verfügung, um Objekte in Datenspeichern zu persistieren. Als Datenspeicher können entweder SQLite, XML oder ein binäres Dateiformat verwendet werden. Die Core Data API abstrahiert vollständig vom zugrundeliegenden Datenspeicher und befreit den Entwickler von direkten Interaktionen mit diesem. Core Data ist aber kein OR-Mapping Framework, sondern ein Objekt-Graph Management Framework. Es verwaltet einen potenziell sehr großen Objektgraphen mit vielen Objektinstanzen und erlaubt es einer Anwendung mit Graphen zu arbeiten, die in Ihrer Gesamtheit nicht in den Speicher passen würden. Dies wird ermöglicht, indem immer nur die Teile des Graphen im Speicher gehalten werden, die gerade benötigt werden. Bei Bedarf werden neue Objekte nachgeladen, oder nicht mehr benötigte wieder ausgelagert. Das Framework behält auch automatisch den Überblick über Änderungen an Objektinstanzen im Objekt-Graph und persistiert diese entsprechend, was den Persistenz Aspekt des Core Data Frameworks ausmacht.

Das zugrundeliegende Schema des Models wird in einem grafischen Editor erstellt und Modifiziert. Dem Entwickler wird somit einen grafische Herangehensweise an das Modeldesign geboten. Eine Eigenschaft, welche die iOS Core Data API von vorhandenen Frameworks auf der Android Plattform abhebt, wo solche eine Arbeitsweise nicht möglich ist.

4.2.3 Datensynchronisation und Konfliktlösungsstrategien

Die Begriffe der Datensynchronisation und damit einhergehend der Replikation von Daten finden ihren Ursprung in dem Umfeld der klassischen verteilten Systeme. Hier wurden unter anderem Konzepte und Verfahren entwickelt, welche die Kommunikation, Replikation und Synchronisation verteilt ablaufender Anwendungen ermöglichen. Im Folgenden soll auf eine Auswahl von Synchronisationsmechanismen eingegangen werden und Ihre Relevanz in Bezug auf die Problemstellung des Mohito Projekts analysiert werden.

4.2.3.1 Replikations- und Synchronisationsverfahren

Wenn verteilte Anwendungen auf gemeinsamen Daten arbeiten sollen, gibt es generell zwei Ansätze dies zu realisieren. Eine Möglichkeit ist auf die Replikation der Daten generell zu verzichten und nur eine zentrale Datenquelle zu haben, auf welcher alle Anwendungen über entsprechende Kommunikationsmechanismen arbeiten. Grundvoraussetzung eines solchen Systems ist allerdings eine hochverfügbare und performante Netzwerkinfrastruktur und die Bedingung, dass alle beteiligten Klienten jederzeit Teil dieses Netzwerks sind. Damit werden allerdings Fälle, in denen sich ein Client im „Disconnected Mode“ befindet nicht unterstützt. Dies stellt aber im Rahmen des Mohito Projekts eine grundlegende Anforderung dar. Hier soll es möglich sein, dass Anwendungen, die sich auf mobilen Clients befinden, auch im Fall einer eingeschränkten oder nicht vorhandenen Netzwerkverbindung weiter fehlerfrei auf den gemeinsamen Daten arbeiten können.

Ein zweiter Ansatz beschreibt verschiedene Verfahren, die gemeinsamen Daten zu replizieren, also auch auf Geräten, die temporär nicht mit dem Netzwerk verbunden sein können, verfügbar zu machen und entsprechend synchron zu halten. Hier wird generell zwischen zwei Arten der Replikations- und Synchronisationsverfahren unterschieden.. Dies sind zum einem die konsistenzerhaltenden, zum anderen die verfügbarkeitserhaltenden Verfahren, welche nachstehend betrachtet werden sollen.

4.2.3.1.1 Konsistenzerhaltende Verfahren

Oberstes Ziel der konsistenzerhaltenden Verfahren ist die Konsistenz innerhalb der Replikationsumgebung. Dabei werden Einschränkungen bei der Verfügbarkeit in Kauf genommen. Solche Verfahren unterteilen sich nochmals in pessimistische und optimistische Verfahren. Bei der pessimistischen Variante werden mögliche Inkonsistenzen durch Sperren (Mutex) im Datenbanksystem von vorherein ausgeschlossen. Optimistische Verfahren hingegen finden Verwendung, wenn angenommen werden kann, dass Inkonsistenzen nur selten auftreten, da beispielsweise hauptsächlich lesend auf die Daten zugegriffen wird. Sie erlauben das temporäre Auftreten von Inkonsistenzen während der Ausführung von Transaktionen, prüfen aber am Ende in einer Validierungsphase, ob Inkonsistenzen vorliegen. Diese müssen dann ent-

sprechend aufgelöst werden, oder die ganze Transaktion muss alternativ zurückgenommen werden. Um dies detaillierter zu veranschaulichen sollen zwei Verfahren vorgestellt werden.

4.2.3.1.1.1 ROWA Verfahren

Das Akronym ROWA steht für „Read One Write All“. Die Kernidee des Verfahrens besteht darin Änderungen an einem Datenobjekt immer synchron auf allen Replikaten durchzuführen. Dazu müssen vor jeder Änderungstransaktion alle zu aktualisierenden Kopien gesperrt werden. Nur dann kann eine Änderung auf allen Replikaten erfolgreich durchgeführt werden. Andernfalls muss die Änderungstransaktion verworfen werden. ROWA garantiert so die vollständige Konsistenz aller Replikate. Das Lesen veralteter oder inkonsistenter Daten ist damit per Design ausgeschlossen. Allerdings ist das ROWA Verfahren stets auf die Erreichbarkeit aller Kopien angewiesen, was gerade bei mobilen Anwendungen nicht gewährleistet werden kann. Um diese Einschränkung abzumildern wurde eine modifizierte Version, das ROWAA („Read One Write All Available“) Verfahren eingeführt. Hier wird die Erreichbarkeit der vorhandenen Kopien berücksichtigt. Es genügt bei dieser Variante alle gerade verfügbaren Datenobjekte zu aktualisieren. Auf den, zu dem Zeitpunkt der Transaktion nicht verfügbaren Replikaten, müssen dann asynchron zu einem späteren Zeitpunkt die entsprechenden Transaktionen nachgeführt werden. Diese Variante des ROWA Verfahrens wäre zwar in einem mobilen Umfeld theoretisch denkbar, ist aber auf Grund des hohen Verwaltungsaufwandes der nicht erreichbaren Klienten und der entsprechenden asynchronen Aktualisierungsmechanismen eher ungeeignet. [20]

4.2.3.1.1.2 Primary Verfahren

Das Primary Verfahren Löst das Problem der Aktualisierung aller Replikate, indem eine Primärkopie oder Masterkopie bestimmt wird. Es ist also ein zentralistischer Ansatz, der die Gleichheit aller beteiligten Replikate zugunsten eines Masterreplikats aufgibt. Bei einer Änderungstransaktion wird nun zuerst die Masterkopie gesperrt und Änderung daran vorgenommen. Sind diese Änderungen erfolgreich, gilt die Transaktion als erfolgreich. Die Masterkopie ist nun in einem zweiten Schritt dafür verantwortlich die Änderungen an alle anderen Replikate zu propagieren, um wieder eine konsistente Replikationsumgebung herzustellen. [20]

Dieses Verfahren hat ROWA gegenüber gewisse Vorteile, da nur noch eine Kopie, nämlich die Primärkopie gesperrt werden muss. Allerdings gerät diese Primärkopie, bzw. der Knoten auf welchem sie liegt, schnell zum Flaschenhals, da hier alle Transaktionen zusammenlaufen.
Mit gewissen Einschränkungen wäre dieses Verfahren auch im mobilen Umfeld einsetzbar, wenn man beispielsweise definieren würde, dass eine Masterkopie niemals auf einem potenziell nicht erreichbaren mobilen Client liegen darf. Dennoch müsste ein beliebiger mobiler Client, der eine Änderungstransaktion initiieren möchte auf die Masterkopie zugreifen kön-

nen, was bedeutet, dass im Offline Fall, keine Änderungen an der Datenbank vorgenommen werden können. Dies ist jedoch im Rahmen der Anforderungen des Mohito Projekts nicht akzeptabel. Allerdings stellt das hier beschriebene Primary Verfahren in gewisser Weise eine Grundlage für ein Synchronisationsverfahren dar, das für die Anforderungen des Mohito Projekts geeignet sein könnte und im Kapitel, dass sich mit geeigneten Verfahren für mobile Umgebungen beschäftigt, noch näher vorgestellt wird.

4.2.3.1.2 Verfügbarkeitserhaltende Verfahren

Im Gegensatz zu den vorhergehend vorgestellten Verfahren, bei denen die Konsistenz der Daten im Vordergrund stand, versuchen die hier erläuterten Verfahren eine möglichst hohe Verfügbarkeit der Daten zu ermöglichen. Da die Konsistenz von Daten in einer Replikationsumgebung jedoch in direktem Widerspruch zu deren Verfügbarkeit steht, bedeutet eine Annäherung an eines der beiden Ziele immer auch die Abkehr von dem jeweils anderen. Das heißt eine hohe Verfügbarkeit lässt sich nur durch eine eingeschränkte Konsistenz erreichen. Im Bereich der verfügbarkeitserhaltenden Verfahren werden daher Inkonsistenzen in bestimmten Situationen und in definierten Grenzen zugelassen. Dadurch entstandene Konflikte müssen dann in einem späteren Schritt in geeigneter Weise aufgelöst werden. Auf entsprechende Konfliktlösungsstrategien soll in einem eigenen Kapitel eingegangen werden. Hier sollen zwei Vertreter verfügbarkeitserhaltender Verfahren vorgestellt und deren Relevanz für mobile Umgebungen erörtert werden.

4.2.3.1.2.1 Epsilon-Serialisierbarkeit

Bei diesem Verfahren wird eine Dateninkonsistenz in gewissen Grenzen zugunsten einer höheren Datenverfügbarkeit geduldet. Die tolerierbare Abweichung von Replikaten gegenüber einem bestimmten Konsistenzzustand wird anhand eines globalen Epsilon Faktors festgelegt. Jedes Replikat ist selbst verantwortlich seine individuelle Abweichung zu dokumentieren und Lesetransaktionen nur innerhalb der erlaubten Abweichungsgrenze zuzulassen. Für jede Lesetransaktion gilt: Solange die Inkonsistenz kleiner als die Überlappungsgrenze Epsilon ist, dürfen sich die Aktionen einer Lesetransaktion beliebig mit den Aktion anderer mit dieser in Konflikt stehenden Transaktionen überschneiden. [20]

4.2.3.1.2.2 Quasi-Copy

Eine Variante der Epsilon-Serialisierbarkeit ist das sogenannte „Quasi-Copy“ Verfahren. Hier wird die tolerierbare Abweichung eines Replikats durch die Anzahl lokal erfolgter Änderungen auf dem Replikat definiert. Übersteigt die Anzahl der Änderungen eine durch Epsilon festgelegte Grenze, dürfen keine weiteren Lesetransaktionen auf dieser Kopie durchgeführt werden. Eine weitere Festlegung von Epsilon kann bei diesem Verfahren auch die Zeitspanne sein, die zwischen zwei Updatetransaktionen liegt.

Durch das Festlegen eines geeigneten Epsilons sind beide vorgestellten Verfahren in mobile Umgebungen einsetzbar, da innerhalb gewisser Grenzen autonomes arbeiten auf einer Kopie möglich ist. Allerdings wird durch die Definition von Epsilon gleichzeitig die Notwendigkeit regelmäßiger Updates bzw. entsprechender Onlinephasen vorgegeben. [21]

4.2.3.1.3 Geeignete Verfahren für mobile Umgebungen

Alle bisher beschriebenen Verfahren sind nur bedingt oder mit gewissen Einschränkungen für den Einsatz in mobilen Umgebungen geeignet. Grund dafür ist, dass alle Verfahren ihren Ursprung bei den klassischen verteilten Systemen haben, in denen der Offline Zustand eines Knotens eine Ausnahme bzw. einen Fehlerfall darstellt. Bei verteilten Systemen, an welchen auch mobile Clients beteiligt sind, repräsentiert ein Knoten, der nicht mit dem Netzwerk verbunden ist, dagegen einen zulässigen Zustand und darf keines Falls als Fehler oder Ausnahme angesehen werden. Daher lassen sich die konventionellen Verfahren nur bedingt an mobile Anforderungen anpassen. Dies trifft vor allem auf die pessimistischen Konsistenzerhaltenden Verfahren zu, da diese zumeist mit Sperren arbeiten, was für mobile Clients fatal sein kann. Wechselt ein solcher Client bei gesetzter Sperre in den nicht-verbundenen Zustand (Disconnected Mode) würde die Sperre unverhältnismäßig lange bestehen bleiben und ein weiteres Arbeiten auf der verteilten Datenbank verhindern. Besser geeignet sind hier die optimistischen oder verfügbarkeitserhaltende Verfahren, aber auch hier müssen entsprechende Probleme und Einschränkungen behandelt werden.

Aufgrund der immer wichtiger werdenden Bedeutung mobiler Datenbanksysteme und der zunehmenden Integration mobiler Clients in bestehende Unternehmensstrukturen, entstanden auch Synchronisationsverfahren, die speziell für mobile Umgebungen geeignet sind. Ein solches Verfahren, das auf dem schon vorgestellten Primary Verfahren aufbaut, soll im Folgenden beschrieben werden.

4.2.3.1.3.1 Virtual-Primary-Copy

Im Gegensatz zu den meisten hier vorgestellten optimistischen oder verfügbarkeitserhaltende Verfahren, die in irgendeiner Form von einem Masterknoten oder einer Primärkopie ausgehen, welche sich immer in einem konsistenten und aktuellen Zustand befindet, wird bei diesem Verfahren eine weitere Abstraktionsebene eingeführt. Es ist nicht mehr ein dedizierter Knoten im System verfügbar, der einen bestimmten Konsistenzzustand repräsentiert, gegenüber dem gewisse Abweichungen erlaubt sind und mit dem Synchronisationsvorgänge stattfinden. Stattdessen geht man von einer virtuellen, Primärkopie aus, deren Zustand zu jeder Zeit in der Replikationsumgebung ermittelbar sein muss. Diese virtuelle Kopie, die eine Abstraktion von einer physikalisch vorhandenen Kopie darstellt, repräsentiert nun den aktuellen Wert eines logischen Datenobjekts.

Da das logische Objekt nur eine Abstraktion darstellt, kann der entsprechende, aktuelle Wert nur ermittelt werden, indem man physikalisch existierende Kopien konsultiert. Dabei ist natürlich die Kopie, die als letzte synchronisiert wurden, diejenige, die den aktuellen Zustand der Replikationsumgebung darstellt. Sie stellt also zum Zeitpunkt x eine valide konsistente Kopie unserer Replikationsumgebung dar. Das heißt von dort ist auch der konsistente Wert unseres logischen Objekts ableitbar. Es kann aber auch sein, dass zu einem Zeitpunkt x mehrere Knoten, konsistente Kopien aufweisen. All diese Knoten bilden dann eine sogenannte Konsistenzinsel (consistency island). Eine Konsistenzinsel kann daher aus einem oder mehreren Knoten bestehen und stellt unsere virtuelle Primäre Kopie dar. Dabei ist es nicht unbedingt nötig, dass alle Replikate der Konsistenzinsel stets den aktuellen Wert repräsentieren, sondern nur, dass diese Replikate gemäß einer synchronen Replikationsstrategie koordiniert werden. Hierzu kann beispielsweise ein Synchronisationsverfahren aus dem Bereich der konsistenzerhaltenden Verfahren genutzt werden, da die Knoten der Konsistenzinsel, zumindest solange sie Bestandteil dieser sind, wie Knoten in konventionellen verteilten Systeme angesehen werden können und damit ständig verfügbar sind. Replikationsstrategien innerhalb einer Konsistenzinsel müssen also nicht den Fall berücksichtigen, dass Knoten nicht mehr erreichbar sind, da sie dann schlicht nicht mehr Bestandteil der Konsistenzinsel sind. Da innerhalb einer Konsistenzinsel die Konsistenz im Mittelpunkt steht werden hier vorzugsweise konsistenzerhaltende Verfahren wie beispielsweise das ROWA Verfahren eingesetzt. [22]

Um zu jedem Zeitpunkt eine valide virtuelle Primärkopie zu haben, muss die Bedingung eingeführt werden, dass die Konsistenzinsel nie verwaisen darf. Es muss also immer mindestens ein konsistentes Replikat in der Konsistenzinsel verbleiben. Das kann beispielsweise dadurch erreicht werden, dass Knoten mit beständigerer und performanterer Netzwerkanbindung an dem Ausscheiden aus der Konsistenzinsel in bestimmten Situationen gehindert werden können.

Replikate, die zu einem bestimmten Zeitpunkt nicht der Konsistenzinsel angehören, werden als Sekundärkopien bezeichnet. Für diese Knoten können nun Konsistenzgarantien nach dem Muster der verfügbarkeitserhaltenden Verfahren definiert werden. So ist für die Synchronisation von Sekundärkopien ein Verfahren aus dem Bereich der Epsilon-Serialisierbarkeit, wie beispielsweise das Quasi-Copy Verfahren denkbar. Über die Wahl eines geeigneten Epsilons, werden die Bedingungen und Grenzen der Autonomie einer Sekundärkopie definiert. Unabhängig von diesen Bedingungen, müssen Sekundärkopien immer wieder mit der virtuellen Primärkopie synchronisiert werden. Dies ist gleichbedeutend mit dem Beitritt zu einer Konsistenzinsel. Geht ein Knoten wieder offline und arbeitet autonom, verlässt er die Konsistenzinsel. [23]

Wenn ein Knoten aus dem Verbund der Sekundärkopien eine geeignete Netzkonnektivität aufweist und sich wieder mit der virtuellen Primär Kopie synchronisieren möchte muss er mindestens eine gültige Referenz zu einem Knoten haben, der Bestandteil der Konsistenzinsel ist, um sich mit diesem Abgleichen zu können. Dazu verwaltet jeder Knoten eine Liste von Replikaten zusammen mit der Zusatzinformation, ob sich das entsprechende Replikat in der Konsistenzinsel befindet, oder nicht. Die Information über neu zu der Konsistenzinsel hinzugekommene oder sie verlassende Knoten muss im System ebenfalls propagiert werden, um die Listen aktuell zu halten. Damit hat jeder einzelne Knoten die nötige Information welchen Knoten er kontaktieren muss um sich synchronisieren zu können. Die Wahl eines geeigneten Knotens, um möglichst geringe Übertragungskosten, und hohe Übertragungsperformanz zu gewährleisten, muss über entsprechende Routingmechanismen sichergestellt werden.

Das hier beschriebene Virtual-Primary-Copy Verfahren eignet sich hervorragend für den Einsatz in mobilen Umgebungen, da es sowohl die Einschränkungen mobiler Clients berücksichtigt, als auch die Anforderungen an Konsistenz und Verfügbarkeit in sich vereint und in weiten Bereichen dynamisch an die Bedürfnisse spezieller Anwendungsszenarien anpassbar gestaltet.

4.2.3.2 Konfliktlösungsstrategien

Bei der Synchronisation von Replikaten kann es, wie in den vorangegangenen Kapiteln erläutert, zu Konflikten zwischen einzelnen Datensätzen kommen, die entsprechend aufgelöst werden müssen, um die Konsistenz einer Replikationsumgebung zu gewährleisten. Strategien, die das Lösen solcher Konflikte ermöglichen, können in zwei Gruppen eingeteilt werden. Hier sind die regelbasierten Prioritätsverfahren und die Mehrheitsverfahren, auch als summierende Verfahren bezeichnet, zu unterscheiden.
Bei den regelbasierten Prioritätsverfahren wird ein Konflikt über definierte statische Regeln aufgelöst. So kann eine Regel beispielweise festlegen, dass im Konfliktfall immer der neuere Datensatz gewinnt. Es kann aber auch eine Prioritätsfolge für bestimmte Replikationsknoten festgelegt werden, die entscheidet dass Datensätze von einer bestimmten Quelle anderen gegenüber bevorzugt werden (Trust your Friends). Auf diese Weise werden Konflikte durch feste Regeln beseitigt. [24]

Stehen mehr als zwei Quellen zur Auswahl kann der Konflikt auch über ein Mehrheitsverfahren aufgelöst werden, wobei derjenige Datensatz gewinnt, der häufiger vorkommt.

Generell ist das automatisierte Auflösen von Konflikten fehlerbehaftet, da nicht immer durch statische Regeln im Vorfeld entschieden werden kann, welcher Datensatz in einer gewissen Situation der richtige ist. Daher existieren auch Ansätze bei denen der Benutzer über geeig-

nete Benutzerschnittstellen bei der Auflösung eines Konflikts involviert wird, da er oft am besten entscheiden kann, wie ein Konflikt zu lösen ist.

4.2.4 Caching

Mobile Geräte haben heutzutage durch UMTS und WLAN gute Datenverbindungsmöglichkeiten. Allerdings sollte dennoch Datenverkehr über mobile Verbindungen als teure Ressource angesehen und als solche von Entwicklern auch behandelt werden. Zum einen aufgrund des monetären Gesichtspunks, da häufig mobile Datentarife teurer als Festnetztarife und oft im Datenvolumen begrenzt sind. Zum anderen, und dies ist fast noch entscheidender, da eine Datenübertragung über die Luftschnittstelle sehr Energie intensiv ist, was die Betriebszeit des Gerätes stark reduziert. Aus diesen Gründen spielt Caching im mobilen Umfeld, gerade in Verbindung mit Datenbanksystemen eine entscheidende Rolle.

Als Cache wird ein schneller Puffer-Speicher bezeichnet, der den Zugriff auf „teure“ Ressourcen beschleunigt. Und teuer sind hier beispielsweise aufwendige Berechnungen, oder langsame –Zugriffe zu verstehen. Im Bereich der Datenbanksysteme haben sich semantische Caching Strategien als sinnvoll erwiesen. Beim semantischen Caching werden zu den gepufferten Daten zusätzliche Informationen gespeichert, die diese Daten beschreiben, sogenannte Metadaten. Bei Datenbankanfragen können so zu den erhaltenen Ergebnisdaten die zugehörigen Anfragen gespeichert werden, welche eine semantische Beschreibung der Antwortdaten darstellen. Bei der Anfragebearbeitung kann dann mithilfe der semantischen Informationen entschieden werden, ob eine Anfrage ganz, oder zumindest teilweise, aus dem Cache beantwortet werden kann. Wird eine identische Anfrage im Cache gefunden, kann diese direkt zurückgeliefert werden. Ist nach semantischer Analyse der im Cache vorhandenen logischen Informationen eine Teilmenge der gestellten Anfrage vorhanden, muss nur noch eine komplementäre Anfrage an die Datenbank gestellt werden, um die Antwort zu vervollständigen. In beiden Fällen wird Bandbreite, die sonst zum Übertragen der vollständigen Antwort benötigt worden wäre, eingespart. [25]

Bei der Diskussion von Caching Strategien ist auch die Betrachtung der eingesetzten Ersetzungsstrategien ein wichtiger Faktor. Die Aufgabe einer Cache Ersetzungsstrategie ist es diejenigen Einträge im Cache zu identifizieren, die als nächstes gelöscht werden können, um Platz für neue Cacheeinträge zu schaffen. Im Fall von semantischem Caching können die vorhandenen semantischen Informationen auch für die Ersetzungsstrategie genutzt werden. Hierzu werden semantisch ähnliche Tupel zu sogenannten semantischen Regionen (semantic regions) zusammengefasst. Diese Regionen werden dynamisch basierend auf den clientseitig gestellten Anfragen definiert. Die in einer Region enthaltenen Daten werden durch eine Formel basierend auf konjunktiv verknüpften Bedingungen beschrieben. Den Regionen wird jeweils ein für die Verdrängungsstrategie entscheidender Relevanz Faktor zugeordnet. Läuft der Cache voll, wird die Region mit dem niedrigsten Wert aus dem Cache verdrängt. Das bedeutet, dass alle Tupel dieser Region aus dem Cache gelöscht werden. [26]

Die Verwaltung dieser semantischen Regionen ist somit die wesentliche Aufgabe des Cachemanagements bei semantischen Caching Strategien. Für die Performance des Caches sind hier vor allem gute Algorithmen für die richtige Wahl zur Teilung oder Vereinigung sich überschneidender Regionen, sowie optimale Größe und Anzahl der Regionen zu finden.

Eine besondere Variante des Caching, die vor allem für mobile Geräte entwickelt wurde, ist das sogenannte „Data Hoarding“ (auch als Prefetching bezeichnet). Es bezeichnet das lokale Vorhalten von Daten für einen späteren Gebrauch. So werden Daten, die während späterer Offline-Phasen benötigt werden, in Situationen mit guter Konnektivität oder an dedizierten Orten, wie Terminals, die über schnelle WLAN Verbindungen verfügen, auf das Gerät geladen. Wichtig ist hierbei die passende Auswahl geeigneter Daten. Normalerweise ist es aufgrund der begrenzten Cachegröße nicht möglich alle Daten vorzuhalten. Daher ist aufgrund des Nutzerverhalten und der Nutzerpräferenzen, durch Heuristiken, oder über das Wissen einer zu erwartenden Offline-Situation, eine passende Datenauswahl zu treffen. Diese Daten ermöglichen dann in Offline-Situation ein autonomes Arbeiten. Gelegentlich können, sofern zwischenzeitlich Konnektivität besteht, protokollierte Informationen zum Nutzerverhalten an die Basisstation übermittelt werden, um in Zukunft eine verbesserte Auswahl der zu replizierenden Daten treffen zu können. Darüber hinaus besteht die Möglichkeit in solchen Situationen neue, aufgrund des Nutzerverhaltens wahrscheinlich benötigte Daten nachzuladen. [20]

Auf mobilen Geräten im Umfeld verteilter Datenbanksysteme sind beide Caching Varianten sinnvoll, so dass eine Kombination aus semantischem Caching und Data Hoarding auf einem Client möglich ist, um eine größtmögliche Flexibilität und Performance zu erreichen.

5 Literaturverzeichnis

[1] „research2guidance," 1 6 2012. [Online]. Available: http://www.research2guidance.com. [Zugriff am 23 8 2012].

[2] „Bitkom," 2011. [Online]. Available: http://www.bitkom.org/de/publikationen/38338_68209.aspx. [Zugriff am 23 8 2012].

[3] „Core J2EE Patterns," Sun, [Online]. Available: http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html. [Zugriff am 24 8 2012].

[4] „Oracle Technology Network- Java," Oracle, [Online]. Available: http://www.oracle.com/technetwork/java/index-jsp-135919.html. [Zugriff am 24 8 2012].

[5] „DataNucleus AccessPlatform," DataNucleus Project, [Online]. Available: http://www.datanucleus.org/products/accessplatform.html. [Zugriff am 28 8 2012].

[6] „Oracle Technology Network- Java," Oracle, [Online]. Available: http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html. [Zugriff am 24 8 2012].

[7] „EclipseLink JPA," Eclipse Foundation, [Online]. Available: http://www.eclipse.org/eclipselink/jpa.php. [Zugriff am 27 8 2012].

[8] „EclipseLink - FAQ," Eclipse Foundation, [Online]. Available: http://wiki.eclipse.org/EclipseLink/FAQ/General. [Zugriff am 27 8 2012].

[9] „Hibernate Community Documentation," JBoss Inc., 9 8 2012. [Online]. Available: http://docs.jboss.org/hibernate/orm/4.1/quickstart/en-US/html. [Zugriff am 28 8 2012].

[10] „Hibernate - Community," JBoss Inc., 7 3 2012. [Online]. Available: https://community.jboss.org/wiki/SupportedDatabases2. [Zugriff am 27 8 2012].

[11] „Microsoft Developer Network," Microsoft, [Online]. Available: http://msdn.microsoft.com/en-us/library/ff626522(v=VS.92).aspx. [Zugriff am 28 8 2012].

[12] „Apple Developer Guide," Apple Inc., [Online]. Available: https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/UserDefaults/Introduction/Introduction.html#//apple_ref/doc/uid/10000059. [Zugriff am 27 8 2012].

[13] „Android Developer Guide," Google Inc., [Online]. Available: http://developer.android.com/guide/topics/data/data-storage.html. [Zugriff am 27 8 2012].

[14] „SQLite - News," SQLite Consortium, 11 6 2012. [Online]. Available: http://www.sqlite.org/news.html. [Zugriff am 29 8 2012].

[15] „SQLite," SQLite Consortium, [Online]. Available: http://www.sqlite.org/. [Zugriff am 30 8 2012].

[16] „SQLite - Distinctive Features Of SQLite," SQLite Consortium, [Online]. Available: http://www.sqlite.org/different.html. [Zugriff am 29 8 2012].

[17] „Wikipedia - SQLite,“ Wikimedia Foundation, Inc., [Online]. Available: http://en.wikipedia.org/wiki/SQLite#Adoption. [Zugriff am 30 8 2012].

[18] „The K virtual machine (KVM),“ Oracle, [Online]. Available: http://java.sun.com/products/cldc/wp/. [Zugriff am 30 8 2012].

[19] M. Fowler, Patterns of enterprise application architecture, Addison-Wesley, 2003.

[20] B. Mutschler und G. Specht, Mobile Datenbanksysteme: Architektur, Implementierung, Konzepte, Springer, 2004.

[21] R. Alonso, D. Barbara und Garcia-Molina, Quasi-Copies: Efficient Data Sharing for Information Retrieval Systems, Berlin: Springer-Verlag, 1988.

[22] R. Lenz, „The "Virtual-Primary-Copy Approach",“ in s *Dept. of Computer Science*, Erlangen.

[23] R. Lenz, Adaptive Datenreplikation in verteilten Systemen, Leipzig: Teubner, 1997.

[24] J. Bleiholder und F. Naumann, „Kurz erklärt: Datenfusion,“ *Datenbank Spektrum,* Bde. %1 von %2DOI 10.1007/s13222-011-0043-9, 2011.

[25] P. Godfrey und J. Gryz, „Answering Queries by Semantic Caches,“ in s *Proc. DEXA*, Springer, 1998, pp. 485 - 498.

[26] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava und M. Tan, „Semantic Data Caching and Replacement,“ in s *In VLDB’96, Proc. of 22th Int. Conf. on Very Large Data Bases*, Mumbai, Morgan Kaufmann, 1996, p. 330 – 341.

