Fakultät Informatik, Elektrotechnik und Informationstechnik
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Studienarbeit Nr. 2348

# Synchronisation von Android-Mobiltelefonen mit Open-Source-Groupware-Lösungen

Martin Thielefeld

| | |
|---|---|
| **Studiengang:** | Informatik |
| **Prüfer:** | Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel |
| **Betreuer:** | Dipl.-Inf. Damian Philipp |
| **begonnen am:** | 24. August 2011 |
| **beendet am:** | 23. Februar 2012 |
| **CR-Klassifikation:** | H.3.5 |

# Inhaltsverzeichnis


**1 Einleitung** **9**

**2 Grundlagen und verwandte Arbeit** **11**
2.1 Protokolle . . . 11
2.1.1 WebDAV . . . 11
2.1.2 CalDAV . . . 12
2.1.3 CardDAV . . . 13
2.1.4 GroupDAV . . . 13
2.1.5 SyncML . . . 14
2.2 Dateiformate . . . 14
2.2.1 VCalendar Dateiformat . . . 14
2.2.2 Das VCard Dateiformat . . . 15
2.2.3 Ical4j . . . 16
2.2.4 Ical4j-Vcard . . . 17
2.3 Datenzugriff unter Android . . . 18
2.3.1 Kalender . . . 18
2.3.2 Adressbuch . . . 20
2.4 Android Sync Adapter . . . 21
2.5 Hypermatix AnDal . . . 22
2.6 Microsoft Active Sync . . . 23
2.7 Funambol . . . 23
2.8 MyPhoneExplorer . . . 23
2.9 Sprite Backup . . . 23
2.10 DMFS CardDAV- und CalDAV-Sync . . . 24
2.11 Google Kalender . . . 24
2.12 Google Contacts . . . 24

**3 Entwurf** **25**
3.1 Systemmodell und Anforderungen . . . 25
3.2 Architektur . . . 25
3.3 Synchronisationskonzept . . . 27
3.3.1 Operationen . . . 27
3.3.2 Erkennung getätigter Operationen . . . 27
3.3.3 Effizientes Aktualisieren des Sync-Zustandes . . . 29
3.3.4 Konflikterkennung und Behandlung . . . 30
Vorgehen bei Konflikten, deren Auflösung nicht eindeutig ist . . . 32
3.3.5 Operationsausführung nach Konfliktbereinigung . . . 32

3.3.6 Initiale Synchronisation . . . 33

**4 Implementierung** **35**
4.1 Synchronisationslogik . . . 35
4.1.1 Darstellung des Sync-Zustandes im Speicher . . . 35
4.1.2 Erweiterung des VCard Funktionsumfanges durch GevilVCardHash . 36
4.1.3 Effizientes Auslesen von Kontakten aus dem Android Adressbuch . . . 36
4.1.4 Schreiben ins Android Adressbuch . . . 38
Einfügen eines Kontaktes ins Android Adressbuch . . . 39
Löschen eines Kontaktes aus dem Adressbuch . . . 39
Update eines Kontaktes im Android Adressbuch . . . 40
4.1.5 Speicherung der UID im Adressbuch und Kalender . . . 41
4.1.6 Erkennung von getätigten Operationen . . . 42
4.1.7 Konflikterkennung . . . 43
4.1.8 Ausführung von Operationen . . . 43
4.1.9 Unterschiede bei der Implementierung für Kalendereinträge . . . 44
4.1.10 Mapping zwischen Android- und Vcalendar- Zeitzonen . . . 44
4.1.11 Mapping zwischen Android- und Vcalendar- Wiederholungsregeln . . . 45
4.1.12 Synchronisation von Remindern . . . 46
4.2 Gevil-Sync Sync Adapter . . . 46
4.3 Bereitstellung zusätzlicher Einstellungsmöglichkeiten . . . 47
4.4 Known Issues . . . 48
4.4.1 Ical4j-Vcard . . . 48
4.4.2 Jdom Android Fork . . . 48
4.4.3 Löschen von Wiederholungsregeln unter Android . . . 49
4.5 Future Work . . . 49
4.5.1 Wiederherstellungsfunktion . . . 49
4.5.2 Performanceoptimierung mittels Android Dirty Flag . . . 49
4.5.3 Hintergrundtask zur Anzeige von Notifications . . . 50

**5 Evaluation** **51**
5.1 Methologie . . . 51
5.2 Performance bei der Ausführung von Operationen . . . 53
5.3 Performance beim Abgleich ohne Änderungen . . . 55
5.4 Verursachter Traffic bei Synchronisation . . . 56
5.5 Performancevergleich mit dem Google Kalender . . . 57

**Literaturverzeichnis** **61**

# Abbildungsverzeichnis


3.1 Architektur von Gevil-Sync . . . . . 26
3.2 Grundlegendes Synchronisations Konzept in Gevil-Sync . . . . . 28
3.3 Ausführung der ermittelten Befehle auf Server und Client . . . . . 33

4.1 Zusammengefasster Kontakt in Android . . . . . 37

5.1 Vorgehensweise des Benchmarks beim Vergleich mit Google . . . . . 53
5.2 Synchronisationsdauer gemischter Operationen . . . . . 54
5.3 Synchronisationsdauer unveränderter Einträge . . . . . 55
5.4 Von der Synchronisation benötigtes Datenvolumen . . . . . 57
5.5 Vergleich mit Google bei drei Operationen . . . . . 58
5.6 Vergleich mit Google, Insert Operationen . . . . . 59
5.7 Vergleich mit Google, Delete Operationen . . . . . 59


# Tabellenverzeichnis


3.1 Erkennung von Operationen, die auf einer der beiden Seiten die synchronisiert werden, getätigt wurden. . . . . 29
3.2 Alle möglichen Konflikte . . . . . 30


# Verzeichnis der Listings


2.1 Authenticator Eintrag in der Manifest Datei . . . . . 22
2.2 Syncadapter.xml . . . . . 22

# Verzeichnis der Algorithmen

4.1 Erkennung getätigter Operationen, SVS und SVH . . . . . 42

## Abstract

In dieser Arbeit werden Grundlagen, der Entwurf und die Implementierung von Gevil-Sync, einer Android Synchronisations App für Adressbuch- und Kalendereinträge beschrieben. Das eigens entworfene Synchronisationskonzept, welches verschiedene Protokolle, darunter GroupDAV, zur Serverkommunikation einsetzt, wird erläutert. Es ermöglicht eine Zwei Wege Synchronisation. Das heißt: Sogar wenn Änderungen auf dem Handy und auf dem Server vorliegen, können diese abgeglichen werden. Nach der Umsetzung dieses Konzeptes in eine App folgt die Evaluation, in der die zur Synchronisation benötigte Zeit sowie Datenvolumen ausgewertet werden.

# 1 Einleitung

Eine komfortable Möglichkeit, personenbezogene Daten elektronisch zu verwalten, bietet sogenannte Personal Information Manager (PIM) Software. Diese Programme ermöglichen die Speicherung von Kontakten, Terminen, Aufgaben und Notizen. Auch das Bearbeiten, Austauschen, Verwalten und Synchronisieren dieser Informationen wird von einigen Implementierungen angeboten. Der Fokus dieser Arbeit liegt beim Synchronisieren personenbezogener Daten mit Android Smartphones.

Auf allen Mobiltelefonen, die auf Google's Android Betriebssystem basieren, wird schon von Haus aus das Adressbuch und der Kalender mit den entsprechenden Google online Diensten synchronisiert. Diese Daten werden dann via Internet direkt auf den zu Google's PIM Web Applikationen gehörenden Servern gespeichert und man kann somit von eigentlich jedem internetfähigen Gerät darauf zugreifen. Doch das ist nicht Alles: Auch Google könnte problemlos auf diese Daten, welche auf ihren Servern liegen, zugreifen. Nach Googles Motto "Don't be evil", sollte man davon ausgehen können, dass Google dies nicht tut. Sicherer ist es jedoch, empfindliche Daten gar nicht mit Google zu synchronisieren, wodurch die Informationen dann jedoch nur noch auf dem Handy einsehbar sind und im Falle eines Datenverlustes keine Kopie mehr vohanden ist.

Im Hinblick darauf wurde in dieser Arbeit die Open-Source-Anwendung Gevil-Sync entwickelt, die es ermöglicht, diese Daten mit einem eigenen GroupDAV Server zu synchronisieren, ohne dass für Google irgendeine Zugriffsmöglichkeit auf die persönlichen Daten entsteht. Ein weiterer Vorteil bei der Verwendung eines eigenen Servers ist, dass Informationen dort komfortabel bearbeitet werden können. Neben der Bearbeitung mit dem Handy können das Adressbuch und der Kalender mit allen Tools modifiziert werden, die den Zugriff auf GroupDAV ermöglichen.

Durch die Verwendung etablierter Protokolle soll es außerdem möglich sein, die synchronisierten Informationen auch auf Handys anderer Hersteller zu bringen, die keine Synchronisierung mit Google Accounts unterstützen.

Im Grundlagenkapitel werden existierende Techniken wie Protokolle, Dateiformate und Anwendungen, die für Gevil-Sync in Frage kommen und teilweise Verwendung fanden, besprochen. Der Entwurf von Algorithmen und der Anwendungsarchitektur wird in Kapitel 3 dargelegt und in Kapitel 4, wird erläutert, wie dieser Entwurf in ein lauffähiges Programm umgesetzt wurde. Schließlich werden in Kapitel 5 Benchmarkergebnisse grafisch dargestellt und Resultate daraus gezogen.

# 2 Grundlagen und verwandte Arbeit

Die Android App Gevil-Sync zählt sich zur Gattung der PIM Software. Hier gibt es bereits sehr viel unterschiedliche Software mit unterschiedlichen Fähigkeiten. Da es für derartige Aufgaben verschiedene Programme gibt, existieren noch mehr unterschiedliche Protokolle, Techniken und Vorgehensweisen. Um Gevil-Sync mit möglichst vielen anderen Anwendungen kompatibel zu machen und um zu vermeiden, dass noch mehr heterogene Protokolle entstehen, werden in diesem Kapitel etablierte und eventuell nützliche Protokolle betrachtet. Neben deren Funktionsweise wird dargelegt, welche Protokolle in Gevil-Sync Einsatz fanden und was die Gründe für oder gegen die Nutzung einer Technik waren.

## 2.1 Protokolle

Zu Beginn werden alle Protokolle deren Verwendung Gevil-Sync in Frage kommt beschrieben.

### 2.1.1 WebDAV

Das WebDAV Protokoll ist eine Erweiterung zu HTTP. Die Abkürzung DAV steht für Distributed Authoring and Versioning. Das Protokoll bietet die Möglichkeit, Daten auf einen Server hochzuladen oder dort zu editieren (Authoring). Dabei ist es möglich, dass mehrere Nutzer von verschiedenen Orten zusammen an den gleichen Daten arbeiten (Distributed). Auch eine Versionskontrolle ist vorgesehen (Versioning). Die Möglichkeit Daten hochzuladen hat HTTP auch ohne WebDAV-Erweiterung per HTTP-Post geboten. WebDAV fügt die folgenden zwei Erweiterungen dazu ein:

1. Die overwrite prevention verhindert, dass eine Datei von einem Nutzer geschrieben wird, solange sie von einem anderen Nutzer bearbeitet wird.
2. Es ist möglich, beliebige Eigenschaften für Dateien zu definieren. Beispiele wären Author, Dateilänge oder Datum der letzten Änderung. Diese Eigenschaften, auch Metadaten genannt, werden im XML Format gespeichert. Wenn eine solche sich durch Verschieben oder Kopieren einer Ressource ändern kann, nennt man sie live Property, zum Beispiel das Datum der letzten Änderung. Andernfalls handelt es sich um eine static Property.

Um diese, durch WebDAV ergänzten Erweiterungen zu realisieren, ist es nötig dass der Client Funktionen, die vom Server angeboten werden, aufruft. Schwierig ist hierbei vor allem die Parameterübergabe. Schon in HTTP war dies über kleine Tricks möglich, jedoch mit Nachteilen und Sicherheitsrisiken verbunden. Aus diesem Grund wurde in WebDAV eine neue Methode namens PROPFIND eingeführt. Im folgenden Beispiel wird ein PROPFIND genutzt, um das Inhaltsverzeichnis eines Dateiverzeichnisses samt Metadaten herunterzuladen [Duso7] [JWJ98].

Im folgenden Beispiel wird ein PROPFIND an einen WebDAV Server, der unter http://192.168.239.128/ erreichbar ist, geschickt: `PROPFIND http://192.168.239.128/Calendar`. Darauf antwortet dieser mit der xml Datei:

```
<?xml version="1.0" encoding="utf-8"?>
<multistatus xmlns="DAV:" xmlns:G="http://groupdav.org/">
       <response>
              <href>http://192.168.239.128/groupdav/Calendar/4eh3f-4</href>
              <propstat>
                     <status>HTTP/1.1 200 OK</status>
                     <prop>
                            <displayname>Calendar</displayname>
                            <resourcetype><collection/><G:vevent-collection /></resourcetype>
                            <getlastmodified>Mon, 29 Aug 2011 17:40:43
                                +0500</getlastmodified>
                     </prop>
              </propstat>
       </response>

       <response>
       ...
       </response>
</multistatus>
```

In der xml Antwort gibt jede `<response>` unter `<href>` (Zeilen drei und vier) den Zugriffspfad auf einen Kalendereintrag preis. Unter jedem `<href>` ist eine VCalendar Datei verlinkt, die einen Eintrag im Kalender repräsentiert. Jeder dieser Einträge kann mit einem HTTP-Get vom Server geladen werden.

### 2.1.2 CalDAV

Das Calendar Sharing and Scheduling Protokoll (CalDAV) benutzt WebDAV, um auf Termine im Ical Format (Abschnitt 2.2.1) zuzugreifen.

Zum Herunterladen, Hochladen und Überschreiben, sowie zum Löschen eines Termines nutzt es die HTTP Standardfunktionen. Zur schnellen Identifikation von veränderten Kalendereinträgen kommt der HTTP Etag zum Einsatz. Dabei handelt es sich um einen ganzzahligen Wert, der vor allem bei der Synchronisation von Daten Verwendung findet. Sobald auf dem Server ein Eintrag, im Falle von CalDAV Termin, verändert wird, ändert sich auch der Etag Wert. Der Client speichert den aktuellen Etag zu jedem heruntergeladenen Datensatz. Soll der Datensatz zu einem späteren Zeitpunkt aktualisiert werden, wird als

erstes der zwischengespeicherte Etag Wert vom Client mit dem, aus der Server Antwort auf das PROPFIND ausgelesenen, verglichen. Nur falls die beiden sich unterscheiden, muss der zugehörige Datensatz überhaupt heruntergeladen werden.

Die von WebDAV eingeführte overwrite prevention wird verwendet, wenn mehrere Nutzer Einträge bearbeiten wollen. Mit WebDAV Copy können Kalender kopiert und mit WebDAV Move verschoben werden.

Die von WebDAV angebotenen Methoden haben Nachteile in der Effizienz, wenn ein Kalender durchsucht werden soll. Um herauszufinden, welche Termine in einem Kalender während eines bestimmten Zeitraumes eingetragen sind, wäre es nötig, den gesamten Kalender herunterzuladen, was sehr zeitaufwendig ist. Deshalb wurde das WebDAV Protokoll zusätzlich um die Time-Range-Report Funktion erweitert. Diese ermöglicht es, mit einer einzigen Anfrage alle Termine eines bestimmten Zeitraumes serverseitig zu ermitteln. Eine interessante Variante dieser Time-Range-Reports sind die anonymen Free-Busy-Queries, die es ermöglichen, dass ein Nutzer einem anderen den Lesezugriff auf seinen Kalender erlaubt, jedoch mit der Einschränkung, dass nur ersichtlich ist, zu welchen Zeitpunkten noch kein Termin eingetragen ist. Auf diese Art soll es erleichtert werden, einen gemeinsamen Zeitpunkt zur Vereinbarung eines neuen Termines zu finden. Auch wiederkehrende Termine, wie zum Beispiel Geburtstage, werden in WebDAV durch eine einheitliche Syntax dargestellt.

Außerdem führt CalDAV Collections ein. Diese werden genutzt, um Kalender verschiedener Personen voneinander abzugrenzen [LD05] [CD07].

### 2.1.3 CardDAV

CalDAV unterstützt keine Adressbucheinträge. CardDAV erweitert WebDAV um diese Funktionalität. Als Dateiformat kommt dabei standardmäßig das vCard Format (Abschnitt 2.2.2) zum Einsatz. Prinzipiell könnte CardDAV aber auch in Verbindung mit einem anderen Format genutzt werden. Jeder CardDAV Server muss Etags unterstützen und eine serverseitige Adressdatensuche ermöglichen, so dass das Herunterladen aller Adressbucheinträge zur Durchführung einer Suche vermieden werden kann [Dab11].

### 2.1.4 GroupDAV

GroupDAV bietet, wie auch CalDAV, die Möglichkeit, auf einen Onlinekalender zuzugreifen. Ein Vorteil gegenüber CalDAV liegt darin, dass in einem einzigen Protokoll die Distributed Authoring and Versioning Funktionen für Kalender- und Adressbucheinträge sowie Notizen angeboten werden. Betrachtet man Kalendereinträge, hat GroupDAV das gleiche grundlegende Modell wie CalDAV, jedoch versucht es sich auf die wesentlichen Funktionen zu beschränken, wodurch eine unkompliziertere Implementierbarkeit auf Clients ermöglicht wird. Die wichtigste Funktion, auf die GroupDAV verzichtet, ist die unter Abschnitt 2.1.2 CalDAV bereits erläuterte, anonyme Free-Busy-Query. GroupDAV kann mit jedem

Server, der WebDAV unterstützt, genutzt werden. Die meisten Server, die CalDAV unterstützen, sprechen auch das WebDAV Protokoll und sind somit implizit auch zu GroupDAV kompatibel [Hes04].

### 2.1.5 SyncML

SyncML ist ein einheitlicher Standard zur Datensynchronisation. Unterstützt wird der Abgleich von E-Mails, Kalendereinträgen, Kontaktinformationen und Dokumenten. Es kann auch leicht die Unterstützung weiterer Formate eingebaut werden. Die Datenrepräsentation findet mit XML statt. Weil das Protokoll unter anderem auf Handys mit einer schlechten Internetanbindung arbeiten soll, werden die Daten zur Übertragung in WBXML, einem binär codierten XML Code, übertragen. SyncML ist unabhängig von den tiefer liegenden Netzwerkschichten, um auf möglichst vielen Geräten funktionieren zu können. [LN01] Die eigentliche Synchronisation funktioniert über sogenannte Sync-anchors. Der Last-anchor stellt den Zustand dar, der nach Abschluss der letzten Synchronisation bestand, der Next-anchor enthält den gegenwärtigen Zustand. Nach Beginn der Synchronisation wird der Last-anchor des Clients mit dem Next-anchor des Servers verglichen. Aus den Unterschieden kann erschlossen werden, welche Dateien ausgetauscht werden müssen. Mögliche Synchronisationsarten sind die Zwei-Wege-Sync, bei der die Daten in beide Richtungen abgeglichen werden, die langsame Synchronisation, bei der Bit für Bit exakt verglichen werden, die Einweg-Synchronisation und die Erneuerungs-Synchronisation, bei der der Inhalt der einen Seite komplett mit dem der anderen überschrieben wird. Das Protokoll findet in Gevil-Sync keinen Einsatz. [Man03]

## 2.2 Dateiformate

Im diesem Abschnitt werden zwei Dateiformate sowie Bibliotheken mit denen diese unter Java genutzt werden können, erläutert.

### 2.2.1 VCalendar Dateiformat

Das VCalendar Dateiformat nutzt zur Darstellung jedes Kalendereintrages eine eigene Datei. Im Rahmen dieser Arbeit werden niemals mehrere Einträge zu einer Datei zusammengefasst. In einem VCalendar findet sich immer eine `componente`, namens `VEVENT`, deren Anfang in der Zeile `BEGIN:VEVENT` und deren Ende in der Zeile `END:VEVENT` zu finden ist. Darin enthalten ist eine Liste von Properties. Einige Properties müssen in jedem gültigen VCalendar enthalten sein, andere sind optional.

Eine beispielhafte VCalender Datei, hier ohne sich wiederholende Termine, sieht folgendermaßen aus:

```
BEGIN:VCALENDAR
PRODID:-//Citadel//NONSGML Citadel Calendar//EN
VERSION:2.0
METHOD:PUBLISH
BEGIN:VEVENT
DTSTAMP:20110831T075717
SUMMARY:Geburtstag Max Mustermann
LOCATION:Stuttgart Kaltental
DESCRIPTION:Geschenkgutschein nicht vergessen.
DTSTART;TZID=UTC:20110809T062000
DTEND;TZID=UTC:20110901T084000
TRANSP:OPAQUE
UID:4e5de95d-a27-0
SEQUENCE:1
ORGANIZER:MAILTO:anonymous@ano.de
END:VEVENT
END:VCALENDAR
```

Jede Zeile innerhalb des `VEVENT` im Beispiel stellt eine Property dar. Zu einigen Properties ist eine ergänzende Liste von Parametern gespeichert. Beispielsweise ist in der Parameterliste der `DTSTART` Property, der Parameter `TZID=UTC` gespeichert, welcher aussagt, dass es sich um einen Uiversal Time Zone Identifier (UTC) handelt. Der Unique Identifier (`UID`) ist in jeder Vcard genau einmal enthalten und identifiziert diese eindeutig. Die Properties `DTSTART` und `DTEND` haben die gleiche Bedeutung wie die gleichnamigen Felder im Android Kalender. Sie geben Start- und Endzeitpunkt des Events an. `DTSTAMP` gibt den Zeitpunkt an, zu dem der Kalendereintrag erstellt wurde. Das VCalendar Format ermöglicht Wiederholungsreln, auf Englisch Recurrence-Rule (RRULE). Zum Beispiel soll ein Geburtstagsevent sich jährlich wiederholen. Um dies in einem VCalendar darzustellen, müsste zusätzlich die Property `RRULE:FREQ=YEARLY` hinzugefügt werden. Durch Ergänzung von `ATTENDEE:MAILTO:Martin Thielefeld` wird dargestellt, dass Martin Thielefeld an diesem Event teilnimmt. Beim VCalendar Format handelt es sich ausschließlich um ein Dateiformat. Es ist daher unabhängig von den darunterliegenden Netzwerkprotokollen. Es kann mit jedem Übertragungsprotokoll kombiniert werden [RHA96]. Seit 1998 hat das VCalendar Format einen Nachfolger mit dem Namen ICalendar. Bis auf einige Details, wie zum Beispiel bei Wiederholungsregeln, ist dieses Format jedoch voll abwärtskompatibel zum VCalendar Format [FD98a] [Fis07].

### 2.2.2 Das VCard Dateiformat

Das VCard Format ähnelt dem VCalendar Format, jedoch stellt es an Stelle von Kalendereinträgen, Adressbucheinträge dar. Außerdem existiert kein `VEVENT` im Dateiformat.

Häufig verwendete Properties sind der Personennamen, Telefonnummern oder Emailadressen. Die zwei letzteren können auch mehr als einmal je VCard vorkommen. Wie auch beim VCalendar kommt für jede Person eine separate Datei zum Einsatz. Vorteil dieser Vorgehensweise ist, dass zum Zugriff, etwa im Zuge einer Synchronisation, nur die nachgefragten Personeninformationen übers Netzwerk transferiert werden müssen, wodurch Daten- und

Akkuverbrauch geschont werden. Eine VCard Datei in der Version 2.1, wie sie in Gevil-Sync Verwendung findet, sieht wie folgt aus:

```
begin:vcard
VERSION:2.1
UID:4e70c94b-9fc-1
VERSION:2.1
n:Rothermel;Kurt;;Prof.,Dr.,rer.,nat.,Dr.,h.,c.,;
fn:Kurt
org;charset=UTF-8:Universität S.
adr;charset=UTF-8:;;;;Stuttgart;Baden-Württemberg;70569;Deutschland
tel;home:1234
tel;work:2345
tel;fax:3456
tel;cell:4567
email;internet:
end:vcard
```

Bei der Person, die darin dargestellt wird, handelt es sich um Professor Rothermel, der an der Universität Stuttgart arbeitet und in Baden-Württemberg wohnt. Vor- und Nachname sowie Namenszusätze sind, in der zur Property `n` gehörenden Parameterliste gespeichert. Um welche Namensinformation es sich handelt, ist durch die Listenposition festgelegt. Damit eine VCard gültig ist, sind mindestens die drei Properties `VERSION`, `n` und `Fn` notwendig. Dabei steht `fn` für den Vornamen. Die verwendete `VERSION` ist 2.1.

Um bei Telefonnummern zu unterscheiden, ob es sich um beispielsweise Handy-, Fax- oder Festnetznummer handelt, kann jeweils ein entsprechender Parameter mitgeführt werden. Bei Emailadressen kann der Typ auf die gleiche Art festgelegt werden [FD98b].

### 2.2.3 Ical4j

Ical4j ist eine Java Bibliothek, die den Umgang mit dem VCalendar Dateiformat implementiert. Der Quellcode des Projekts ist neben der Programmiersprache Java auch für Groovy verfügbar. In Gevil-Sync repräsentiert jedes `Calendar` Objekt einen Kalendereintrag. Ein solches kann entweder aus einem String geparst oder aus einzelnen Properties zusammengefügt werden.

Sei `myCalendarString` ein String, der einen VCalendar enthält. Die `build` Methode benötigt ein Reader Objekt, wodurch ein VCalendar aus unterschiedlichen Quellen wie einem Datenträger gelesen werden kann. In unserem Fall wollen wir ihn aus einem String parsen und verwenden deshalb einen StringReader.

```
CalendarBuilder builder = new CalendarBuilder();
mCalendar = builder.build(new StringReader(myCalendarString));
```

Als nächstes wird gezeigt, wie durch Einfügen von Properties ein VCalendar Objekt zusammengestellt werden kann. Unabhängig von der Art der Property, ist das Vorgehen beim Hinzufügen gleich. Soll zum Beispiel ein Location Property eingefügt werden, brauchen wir zunächst das entsprechende `VEVENT Component` Objekt, welches im folgenden Beispiel

ev genannt wird. Bei mCalendar handelt es sich um ein Calendar Objekt. Eine Eigenheit der Bibliothek ist, dass immer manuelle Casts, hier nach Component, notwendig waren:

```
ComponentList comp = mCalendar.getComponents();
Component ev = (Component)comp.get(0);
ev.getProperties().add(new Location(newLoc));
```

Auch das Hinzufügen neuer Parameter funktioniert für alle Parameter ähnlich. Ein europäisches TzId Property wird beispielsweise so zu einem bestehenden DtStart Property hinzugefügt: `getProp("DtStart").getParameters().add(new TzId("Europe"));`

Befindet sich im Speicher bereits ein Calendar Objekt, dessen Ort wir auslesen wollen, übergeben wir der getProperty() Methode einfach den String „LOCATION“. Für alle anderen Properties, wie zum Beispiel „DESCRIPTION“ oder „DTSTART“, ist das Vorgehen auch hier identisch.

```
ComponentList comp = mCalendar.getComponents();
Property property = ((Component)comp.get(0)).getProperty("LOCATION");
```

Auf Parameter wird mittels der Methode getParameter(), welche in jedem Property vorhanden ist, zugegriffen. Auch hier wird ein String, der die Bezeichnung des gewünschten Parameters enthält, übergeben.

### 2.2.4 Ical4j-Vcard

Ical4j-vcard ist eine Erweiterung der Bibliothek Ical4j. Sie kann in ein Projekt eingebunden werden, welchem sie dann das Lesen, Schreiben sowie Erstellen und Modifizieren von VCards (Abschnitt 2.2.2) ermöglicht [Mod11].

Das Einlesen einer VCard aus einem String geschieht durch Aufruf gleichnamiger Methoden wie beim Einlesen eines VCalendar mit ical4j (Abschnitt 2.2.3). Falls keine VCard vorhanden ist, sondern lediglich einzelne Properties zu einer VCard zusammengebaut werden sollen, wird anders vorgegangen. Zur Illustration wollen wir Herrn Max Mustermann außer Dienst, der die Spitznamen Musterle und Beispielchen trägt, in unsere VCard eintragen. N ist hier das Kürzel für Name, FN für den Vornamen.

```
Property vorName = new net.fortuna.ical4j.vcard.property.FN("Max");
card.getProperties().add(vorName);

List<Parameter> N;
Property name = new net.fortuna.ical4j.vcard.property.N("Mustermann", "Max", new String[]
    {"Musterle", "Beispielchen"}, new String[] {"Herr"}, new String[] {"außer Dienst"});
card.getProperties().add(name);
```

Wie man im Beispiel erkennen kann, sind für Spitznamen, Präfixe und Suffixe String Arrays vorgesehen, welche in der VCard als Liste von Parametern mitgeführt werden. Mehrere Spitznamen sind im Deutschen keine Seltenheit, die Nutzung mehrerer Suffixe kommt bei uns jedoch eher weniger zum Einsatz. In einigen Ländern, zum Beispiel China, gibt es im Gegensatz dazu öfters Namenssufixe. Im Property Package von Ical4j-vcard sind 41

verschiedene Klassen zur Darstellung von Properties vorhanden, darunter auch Telefon und Email. Da der Unterschied beim Arbeiten mit diesen lediglich im Ändern des Klassennamens liegt, werden diese hier nicht vorgestellt.

Soll eine Property aus einer VCard ausgelesen werden, wird dies nicht wie bei Nutzung von Ical4j durch einen String, sondern die `Id` Klasse identifiziert. Im Beispiel haben wir uns für das Lesen des Vornamens, welcher durch ein `fn` Property dargestellt ist, entschieden. `String vorName = mCard.getProperty(Id.FN).getValue();` Wollen wir stattdessen auf die zur Property gehörende Liste von Parametern zugreifen, nutzen wir die Methode `getParameters()` des Property Objekts.

Eine weitere Funktion, die hier illustriert werden soll, ist das Ausgeben einer VCard.

```
VCardOutputter outputter = new VCardOutputter();
StringWriter sw = new StringWriter();
outputter.output(card, sw);
```

In diesem Beispiel wird die VCard von einem `VCardOutputter` in einen `StringWriter` geschrieben. Möglich ist jede Art von Writer, zum Beispiel ein `FileWriter`, um eine VCard auf einem Datenträger zu speichern. [ica11]

## 2.3 Datenzugriff unter Android

Das Android Betriebssystem bietet mehrere Möglichkeiten, Daten dauerhaft zu speichern. Für das Adressbuch und den Kalender kommen sogenannte Content Provider zum Einsatz, welche auf SQLite Datenbanken basieren. Beim Zugriff muss man in SQL Statements denken; syntaktisch verwendet man beim Auslesen von Informationen Cursor Objekte.

### 2.3.1 Kalender

Google bietet für den Kalenderzugriff unter Android eine API, für die aber eine funktionierende Internetverbindung vorausgesetzt wird. Vorgesehen ist, dass nicht auf den Handy-eigenen Kalender, sondern auf den Online Dienst Google-Kalender, mit dem in den Standardeinstellungen automatisch alle Kalendereinträge synchronisiert werden, zugegriffen wird. Diese von Google vorgesehene Vorgehensweise macht neben den zwei Nachteilen, dass dazu Internetvolumen verbraucht wird und das Passwort sowie Login des Google Accounts nötig sind, für das Ziel von Gevil-Sync, die privaten Daten des Nutzers eben nicht mit Google zu teilen, keinen Sinn. Für den Zugriff auf den im Handyspeicher befindlichen Kalender, hat Google in der genutzten Android Version 2.33 gar keine API Dokumentation vorgesehen. Deshalb mussten die Zugriffsdetails experimentell herausgefunden werden. Erschwerend kommt hinzu, dass der Kalender ausschließlich auf einem realen Handy getestet werden kann, weil im so genannten Virtual Device, dem Android Simulator, kein Kalender existiert.

Google speichert alle kalenderspezifischen Daten in der SQLite Datenbank des entsprechenden Content Providers. Die verschiedenen Tabellen werden über URIs adressiert. Mangels

einer einheitlichen Zugriffs API, können diese URIs sowie theoretisch der gesamte Zugriffsvorgang von Gerät zu Gerät variieren. Es hat sich jedoch bislang gezeigt, dass die nötigen Befehle normalerweise identisch sind. Ein nicht sehr gravierender Unterschied besteht darin, dass bei Geräten die auf einer Android Version vor 1.6 basieren, die zum Zugriff notwendige URL eine andere ist [Bla]. Auf einem HTC Desire HD mit Android Version 2.3.3 wurde der Zugriff mit folgenden Ergebnissen erfolgreich getestet: Die einzelnen Events, also Termine, sind in einer Tabelle die via `content://com.android.calendar/events` angesprochen werden kann, abgespeichert. Im Normalfall sind je Gerät mehrere Kalender vorhanden. Deren Spalten `displayname` und `_id` waren im Testgerät unter der URI `content://com.android.calendar/calendars` zu finden. Die Zuordnung der Events zu den einzelnen Kalendern konnte anhand eines aus dem SQL Jargon bekannten Join über die, in beiden Tabellen vorhandene Spalte `_id`, erfolgen. Soll die Event Tabelle ausgelesen werden, muss man dazu die entsprechende Query im Hinterkopf haben, zum Beispiel:

```
SELECT title, dtstart, dtend, delted
FROM content://com.android.calendar/events
```

wird für den Content Provider Zugriff wie folgt realisiert:

```
cursor = contentResolver.query(Uri.parse("content://com.android.calendar/events"),
new String[]{"title", "dtstart", "dtend", "delted"}, null, null, null);
```

Mit diesem Cursor können nun einzelne Termine ausgelesen werden [LD]. In der Android Datenbank sind viele Felder gespeichert, von denen hier nur die wichtigsten betrachtet werden. Diese Felder eines Geburtstags sehen so aus:

```
rrule=FREQ=YEARLY;WKST=SU
BYMONTHDAY=6,BYMONTH=12
hasAlarm=1
timezone=GMT
dtstart=492766000000
reminder_duration=20
_sync_account=PC Sync
dtend=597452340000
allDay=0
delted=0
alerts_vibrate=1
eventTimezone=Europe/Berlin
_sync_account_type=com.htc.pcsc
selected=1
title=Geburtstag Sigrid
_id=1
calendar_id=1
suggest_text_1=Geburtstag Sigrid
duration=P86340S
```

Die `rrule` gibt die Wiederholungsregel an. Der Geburtstag wiederholt sich jährlich am gleichen Tag. `Dtstart` gibt den Startzeitpunkt, `Dtend` den Endzeitpunkt des Geburtstages in Millisekunden an. Ansonsten spielt noch das `title` Feld eine wichtige Rolle.

Für das Eintragen eines Events in den Kalender wird zunächst ein `ContentValues` Objekt erzeugt, in das die Eventdetails eingetragen werden.

```
ContentValues cv = new ContentValues();
cv.put("calendar_id", CalendarID);
cv.put("title", "Geburtstag Sigrid");
cv.put("dtstart", 492766000000);
cv.put("dtend", 597452340000);
```

Die `calendar_id` gibt an, zu welchem Kalender dieses Event gehören soll. Durch das Hinzufügen zum `ContentValues` Objekt können alle für den Kalendereintrag nötigen Felder, wie zum Beispiel die aus dem Geburtstagsbeispiel, definiert werden. Der finale Befehl ist das Einfügen des Events, was mittels eines Insert Befehls geschieht. Dieses sieht in Java so aus:

```
Uri eventsUri = Uri.parse("content://com.android.calendar/calendars");
getContentResolver().insert(eventsUri, cv);
```

Auch hier muss wieder beachtet werden, dass die korrekte URI für den Zugriff gewählt wird.

Auch das Adressbuch hat Google auf Basis einer SQLite Datenbank abgelegt. Trotzdem unterscheiden sich, neben den URIs, auch die notwendigen Zugriffsbefehle weitestgehend.

### 2.3.2 Adressbuch

Jeder Eintrag im Adressbuch hat einen eindeutigen Identifier, der unter der Spalte `_id` der Adresstabelle gespeichert ist. Folgende While-Schleife liest der Reihe nach die Identifier aller auf einem Handy gespeicherten Kontakte aus. Ähnlich wie beim Zugriff auf den Kalender, wird dazu ein Cursor verwendet, der mittels eines SQL Statement ähnlichen Befehls, namens `query`, beschrieben wird:

```
Cursor cur = resolver.query(ContactsContract.Contacts.CONTENT_URI, null, null, null, null);

while (cur.moveToNext()) {
       String id = cur.getString(cur.getColumnIndex(ContactsContract.Contacts._ID));
}
```

Via eines so ausgelesenen Identifiers kann nun an Hand einer weiteren Query ein neuer Cursor c, der eine Kontakt-Tabelle enthält, aus dem Kontakte Content Provider ausgelesen werden. Zu beachten ist, dass es sich beim Cursor c um einen einzelnen Kontakt handelt, der nun weiter verarbeitet wird.

```
while (c.moveToNext()) {
       String mimeType = c.getString(DataQuery.COLUMN_MIMETYPE);

       if(mimeType.equals(ContactsContract.CommonDataKinds.StructuredName.CONTENT_ITEM_TYPE)){
              final String nachName = c.getString(DataQuery.COLUMN_FAMILY_NAME);
              final String vorName = c.getString(DataQuery.COLUMN_GIVEN_NAME);

       }else if(mimeType.equals(ContactsContract.CommonDataKinds.Phone.CONTENT_ITEM_TYPE)){
              final int type = c.getInt(DataQuery.COLUMN_PHONE_TYPE);
```

```
                String phoneNr = c.getString(DataQuery.COLUMN_PHONE_NUMBER);

        }else{
                //Behandlung weiterer MIMETYPES

        }

}
```

Jeder Eintrag in der Tabelle eines einzelnen Kontaktes entspricht einen MIMETYPE. Durch die gezeigte Fallunterscheidung kann somit entschieden werden, um welche Art von Property es sich handelt und entsprechende Informationen ausgelesen werden. Für jede Telefonnummer und Emailadresse kann zusätzlich der Typ ausgelesen werden, welcher durch eine Ganzzahl repräsentiert wird [devb].

## 2.4 Android Sync Adapter

Im Android Einstellungen Menü, unter dem Punkt „Konten und Synchronisation“, finden sich normalerweise in jedem Android Handy ein oder mehre Konten. Sobald man zum Beispiel einen Email Account einrichtet, wird sofort ein entsprechendes Konto bereitgestellt und, vorausgesetzt die Synchronisation ist aktiviert, neue Emails automatisch abgerufen. In diesem Konto wird nicht nur der Nutzername, sondern auch das dazugehörige Passwort gespeichert und jede Anwendung, die die nötigen Berechtigungen mitbringt, kann auf die Anmeldeinformationen sämtlicher Konten zugreifen und sich somit auch dort anmelden. Auch die Google eigene Kalender- und Kontakte Synchronisation nutzt diese Konten zur zentralen Speicherung von Anmeldeinformationen.

Beim Sync Adapter handelt es sich um eine Komponente eines solchen Kontos, die lediglich notwendig ist, wenn eine automatische Synchronisationsausführung gewünscht ist, welche die Synchronisation der Daten im Handy und der Daten im Online-Konto zeitgesteuert automatisch einläutet. Allgemein zugreifbare Daten werden in Android generell in sogenannten Content Providern gespeichert, als Beispiele seien das Android Adressbuch oder der Android Kalender (siehe Abschnitt 2.3.2) genannt. Zur Erstellung eines Sync Adapters ist es obligatorisch, den Content Provider anzugeben, mit dem die Daten abgeglichen werden. Ein Sync Adapter zur Synchronisation von zum Beispiel Daten auf der SD Karte ins Internet, ist daher eigentlich gar nicht möglich. Eine weitere Komponente, die für einen funktionierenden Sync Adapter benötigt wird, ist ein Synchronisations Service, der die Daten im Hintergrund abgleicht. Auch eine GUI zur Abfrage der Anmeldeinformationen wird benötigt. Diese wird lediglich beim ersten Login und falls sich das Passwort geändert hat, angezeigt. Die automatisch startende Synchronisationslogik des Sync Adapters wird in der Methode `onPerformSync()`, welche als Implementierung der gleichnamigen abstrakten Methode der Klasse `AbstractThreadedSyncAdapter` angegeben wird, ausprogrammiert [Ste10].

Damit der Sync Adapter schließlich vom Android Betriebssystem auch als solcher erkannt wird, sind einige Einträge in verschiedenen xml Dateien notwendig.

In der Manifest Datei werden, neben dem Eintrag für die Anmeldeactivity, weitere Einträge benötigt. Diese sollen im Folgenden an einem Beispiel gezeigt werden.

```
<service
2       android:name=".authenticator.AuthenticationService"
        android:exported="true">
4               <intent-filter>
                        <action android:name="android.accounts.AccountAuthenticator" />
6
                </intent-filter>
8               <meta-data
                        android:name="android.accounts.AccountAuthenticator"
10                      android:resource="@xml/authenticator"

12              />
</service>
```

**Listing 2.1:** Authenticator Eintrag in der Manifest Datei

Der in Listing 2.1 dargestellte Eintrag teilt dem Handybetriebssystem mit, dass ein `Authenticator` vorhanden ist. Dieser hat, wie schon beschrieben, die Aufgabe, Anmeldedaten zu speichern, sowie Details der Anmeldung am dazugehörigen Server zu implementieren. Zeile 2 in Listing 2.1 gibt die Klasse an, in der der beschriebene Service implementiert ist. Mit Zeile 10 wird auf eine weitere XML Datei verwiesen, in der Icon und Account-Typ des Authenticators angegeben werden. Ein weiterer Service Eintrag in der Manifest Datei meldet den Sync Adapter beim Betriebssystem an und zeigt auf die Syncadapter.xml, die in Listing 2.2 dargestellt ist:

```
<sync-adapter xmlns:android="http://schemas.android.com/apk/res/android"
2       android:contentAuthority="com.android.contacts"
        android:accountType="com.Gevil"
4       android:supportsUploading="false"
/>
```

**Listing 2.2:** Syncadapter.xml

Zeile 2 in Listing 2.2 gibt den Content Provider an. Im Beispiel handelt es sich um den Kontakte Content Provider und somit um einen Sync Adapter, der Kontakteinträge synchronisiert. Der mit dem Sync Adapter verbundene `Authenticator` wird in Zeile 3 angegeben [devb]. Zu beachten ist hierbei, dass lediglich das Vater Package, in dem der Authenticator enthalten ist, nicht jedoch das Authenticator Package selbst, angegeben wird. Zeile 4 gibt an, ob die vom Sync Adapter synchronisierten Einträge auf dem Handy verändert werden dürfen. Zum Beispiel Facebook Freundes-Einträge können zwar heruntergeladen, nicht jedoch verändert und wieder hochgeladen werden.

## 2.5 Hypermatix AnDal

Hypermatix AnDal synchronisiert Kalender unter Android mit einem CalDAV Server. Nachteilhaft ist, dass es nicht den Android eigenen Kalender verwendet, sondern einen neuen

Kalender zur Verfügung stellt, sowie, dass neben Kalendereinträgen keine Informationen synchronisiert werden können. Der Vorgänger des Programms ist Hypermatix Calendar Sync for Android. Die Entwicklung des Vorgängers wurde eingestellt [AnD].

## 2.6 Microsoft Active Sync

Microsofts Active Sync Protokoll unterstützt die Synchronisation von E-Mails, Kontakten, Kalendereinträgen und Notizen. Das gleichnamige Programm synchronisiert Microsoft Handys mit zum Beispiel Outlook. Ein Client, der Active Sync unterstützt, ist das Iphone. Daten die mit Gevil-Sync und einem entsprechenden Server synchronisiert wurden, können über Active Sync auf ein Microsoft Handy übertragen werden. Auf die gleiche Art wäre es möglich, Daten zwischen Android und Iphones zu synchronisieren [App].

## 2.7 Funambol

Funambol ist ein Server, der auf SyncML basiert. Er unterstützt die Synchronisation von Kalendern, Adressen und Emails. Die Software ist unter Windows und Linux lauffähig. Die Datensynchronisation kann mit iPods, Palms und Blackberrys via Plugin stattfinden. Emails können über einen Push Dienst empfangen werden [joo8].

## 2.8 MyPhoneExplorer

Ermöglicht auf Sony Ericsson und Android Handys die Synchronisation von Kontakten und dem Kalender mit Google, Mozilla Thunderbird und Outlook. Außerdem bietet es viele weitere Funktionen wie das Verfassen und Versenden von SMS oder Zugriff und Verwaltung von Daten auf der SD Karte. [Wec11]

## 2.9 Sprite Backup

Sprite Backup ist kein Synchronisationstool, sondern konzentriert sich auf vollständige Backups. Die Software ist kostenpflichtig und neben Android auch für Windows Mobile erhältlich. Gesichert werden können mit dem Programm Kontakte, Kalender, System Einstellungen, SMS, Bookmarks, Fotos und mehr. Als Ziel der Sicherung kann zwischen der SD Karte und Online Diensten wie Dropbox gewählt werden [NS].

## 2.10 DMFS CardDAV- und CalDAV-Sync

Die beiden, teilweise kostenpflichtigen, Synchronisationstools von DMFS ermöglichen Kalendersynchronisation mit dem Android Kalender auf Basis des CalDAV Protokolls und Kontaktesynchronisation mit dem Android Kontakte Content Provider nach den Regeln des CardDAV Protokolls. Ein Hauptunterschied zu Gevil-Sync ist das von ihm genutzte GroupDAV Protokoll. Ansonsten haben Gevil-Sync und die beiden Lösungen von DMFS teilweise gleiche Fähigkeiten, beispielsweise ermöglichen beide eine Zwei-Wege-Synchronisation. CardDAV-Sync setzt, im Gegensatz zur fein granularen Konfliktbehandlung von Gevil-Sync, in mehr Situationen die Optionen Server Wins beziehungsweise Phone Wins ein. Funktionen der beiden Synchronisations Apps von DMFS, die Gevil-Sync nicht implementiert, sind zum Beispiel die in der Pro Version enthaltene Synchronisation von Webseiten, Firmen und Notizen. Zusätzlich beherrscht die Pro Version die Synchronisation von Geburtstagen, Jahrestagen und Nicknames [DMF11]. Auch Gevil-Sync beherrscht die Synchronisation von Remindern, was zur Synchronisation von Geburtstagen und Jahrestagen identisch ist.

## 2.11 Google Kalender

Google bietet auch einen eigenen Kalender online Service an. Per Sync Adapter können alle Termine automatisch mit dem Google Dienst synchronisiert werden und somit neben der Anzeige im Android Kalender auch in der Web Oberfläche eingesehen und editiert werden. Der dazugehörige Sync Adapter bietet, wie auch Gevil-Sync, die Möglichkeit des Uploads, das heißt Termine können auf dem Handy editiert werden und es findet ein Abgleich mit dem Server statt. Einige Sync Adapter erlauben es nicht, die Daten auf dem Handy zu verändern, ein Beispiel hierfür sind Facebook Kontakte. Seit Juli 2008 funktioniert der Google Kalender auch mit CalDAV [Goo].

## 2.12 Google Contacts

Ähnlich zur Kalender-app, bietet Google auch den Onlinedienst Google Contacts. Auch dieser gleicht zugehörige Kontakte über einen Android Sync Adapter mit dem Adressbuch ab und ermöglicht das Einsehen der Online Kontakte in einer Weboberfläche.

# 3 Entwurf

## 3.1 Systemmodell und Anforderungen

In diesem Abschnitt wird das Systemmodell sowie funktionale und nicht funktionale Anforderungen definiert.

Das Programm soll auf einem Android Smartphone mit Android Version 2.33 lauffähig sein. Es basiert auf der Kommunikation mit einem GroupDAV Server, welcher über eine Internetverbindung erreichbar ist. Ein Server, mit dem die Zusammenarbeit gelingen soll, ist Citadel. Auf dem verwendeten Server muss mindestens ein Benutzerkonto existieren, dessen Anmeldeinformationen dem Programm mitgeteilt werden.

Es soll die Fähigkeit haben, Einträge aus dem Adressbuch, sowie Einträge aus dem Kalender zwischen einem GroupDAV Server und Android zu synchronisieren. Veränderungen von Einträgen auf dem Server und auf dem Handy sollen abgeglichen werden können. Dabei können auch Situationen vorkommen, in denen derselbe Eintrag auf dem Server und dem Handy unterschiedlich verändert beziehungsweise gelöscht wird. In solch einer Situation sollen alle Änderungen erkannt und synchronisiert werden. Nach Synchronisationsabschluss darf keine Änderung verloren gegangen sein und alle Einträge müssen auf Server und Handy identisch sein. Die Ausführung der Synchronisation muss nicht manuell gestartet werden.

Der Synchronisationsvorgang darf den Benutzer bei keiner Tätigkeit, ausgenommen beim Einrichten und Warten der Anwendung, durch Anzeige jedweder Informationen ablenken. Er soll außerdem in einem akzeptablen Zeitrahmen ablaufen, wonach die genutzten Hardware-Ressourcen wieder für andere Aufgaben freigegeben werden. Das für jeden Synchronisationsvorgang verbrauchte Datenvolumen und die verbrauchte Energie soll zur Anzahl synchronisierter Einträge angemessen sein.

## 3.2 Architektur

Abbildung 3.1 zeigt die Architektur von Gevil-Sync. Die Synchronisation beginnt für Kontakte sowie für Kalendereinträge beim GroupDAV Server, mit dem übers Internet kommuniziert wird. Mit der Internetanbindung beginnt die eigentliche Architektur von Gevil-Sync, der Kommunikationsschnittpunkt ist WebDAV. Dort werden alle zur Kommunikation notwendigen Http Methoden, einschließlich WebDAV Erweiterungen zur Verfügung gestellt. GroupDAV setzt, wie bei einem Schichtenmodell, auf WebDAV auf und nutzt dessen angebotene

Methoden, indem zum Beispiel ein PROPFIND ausgeführt wird, die XML Antwort ausgelesen und die darin vorkommenden Card-Dateien heruntergeladen werden. Gevil-Sync nutzt zum Parsen der xml Antowort auf das PROPFIND den freien XML Parser Jdom [Ull].

Von Gevil-Sync her betrachtet stellt die GroupDAV Klasse eine Sicht auf den verwendeten Server her, die das Herunterladen, das Hochladen, sowie das Löschen von VCard beziehungsweise VCalendar Dateien ermöglicht. Die gleiche Sicht auf die Android Content Provider Inhalte wird durch die Klassen `ContactOperations` und `CalendarOperations` hergestellt.

Durch Verwendung der von diesen Klassen erzeugten Sichten, kann der zentrale Sync-Adapter, in dem der Synchronisationsvorgang abläuft, flexibel auf alle notwendigen Informationen zugreifen.

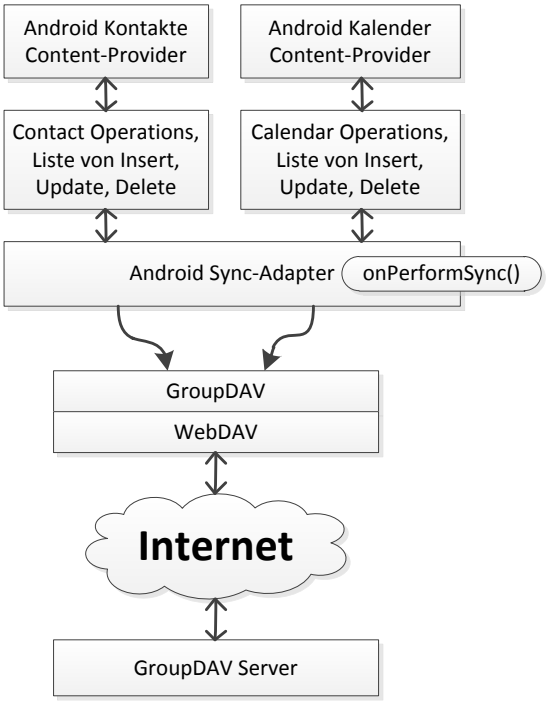


**Abbildung 3.1:** Architektur von Gevil-Sync

Dieser lädt zuerst alle abzugleichenden Informationen aus ihren Quellen und findet notwendige Operationen, welche er nach Konfliktbereinigung an entsprechender Stelle ausführt.

Des Weiteren kann man in der Architektur zwischen der Kontakte- und der Kalendersynchronisation unterscheiden. Architektonisch und algorithmisch existieren zwischen diesen beiden Synchronisationstypen keine nennenswerten Unterschiede; Änderungen finden sich

eher in der Implementierung. Da Adressbucheinträge und Kalenderevents konzeptionell gleich behandelt werden, werden die beiden im Folgenden unter dem Begriff Eintrag vereinheitlicht.

## 3.3 Synchronisationskonzept

Im Synchronisationskonzept wird die Funktionsweise der Synchronisationslogik von Gevil-Sync erläutert und unter Verwendung einiger Grafiken unterstrichen.

### 3.3.1 Operationen

Gevil-Sync bietet eine Zwei-Wege Synchronisation. Einerseits erkennt es, welche Änderungen (=Operationen) auf dem Handy ausgeführt wurden und führt diese auf dem Server aus. Andererseits findet es heraus, welche Operationen auf dem Server getätigt wurden und führt diese auf dem Handy aus.

Im Hinblick darauf, dass alle Einträge unter Android in, auf einer SQLite Datenbank basierenden, Content Providern abgelegt sind, ist die folgende, genauere Definition der Operationen, die erkannt und ausgeführt werden können, naheliegend. Es handelt sich um die drei Befehle Insert, Update und Delete. Diese sind zwar an die gleichnamigen SQL Befehle angelehnt, überschneiden sich semantisch jedoch nur teilweise mit diesen. Das Insert gleicht dem Anlegen eines neuen Eintrages, das Update der Änderung einer oder mehrerer Datenfelder eines bestehenden Eintrages und ein Delete wird erkannt, wenn ein kompletter Eintrag samt aller darin befindlichen Informationen vom Benutzer gelöscht wurde. Jede vom Benutzer durchgeführte Änderung wird bei Synchronisationsausführung von Gevil-Sync erkannt, zu einer dieser drei Operationen zugeordnet und in einer Liste gespeichert. Nach weiteren Schritten werden alle Operationen auf Server beziehungsweise Client ausgeführt, um die Synchronisation zu vervollständigen.

Ein spezielles Augenmerk ist hierbei auf die Granularität der Operationen zu legen. Wird beispielsweise die Emailadresse eines bestehenden Kontaktes gelöscht, gleicht dies keiner Delete Operation, sondern einem Update: Die aktualisierte VCard unterscheidet sich durch das Fehlen einer Emailadresse. Ein Delete tritt lediglich dann auf, wenn ein kompletter Kontakt gelöscht wurde.

### 3.3.2 Erkennung getätigter Operationen

Wenn der Benutzer auf dem Handy einen Eintrag löscht, ist für ein Synchronisationsprogramm ohne vorherigen Zustand nicht erkennbar, ob dieser auf dem Handy gelöscht oder auf dem Server neu erstellt wurde.

Um in diesem und anderen Fällen eindeutig nachvollziehen zu können, was der Benutzer getan hat, speichert Gevil-Sync einen Sync-Zustand. Nach erfolgreichem Abschluss einer

Synchronisation sind alle Informationen, die auf beiden Seiten auf unterschiedliche Art im Speicher dargestellt werden, identisch. Der Sync-Zustand ist eine Kopie dieser beiden identischen Zustände.

In Abbildung 3.2 ist das grundlegende Prinzip der Erkennung getätigter Operationen dargestellt. Die mittlere der drei Datenbanken stellt den Sync-Zustand dar. Die linke Datenbank stellt den momentanen Zustand im Handy dar. Dieser ändert sich immer, wenn der Benutzer oder auch eine dritte Anwendung, Änderungen am Adressbuch beziehungsweise Kalender, vornehmen. Die rechte Datenbank stellt den Zustand dar, der aktuell auf dem Server vorgefunden wird. Auch dieser ändert sich mit jeder Operation, welche auf dem Server getätigt wird. An dieser Stelle soll davon ausgegangen werden, dass in der Vergangenheit schon einmal eine Synchronisation stattgefunden hat. Wie bei der allerersten Synchronisation vorgegangen wird, wird im Anschluss betrachtet.

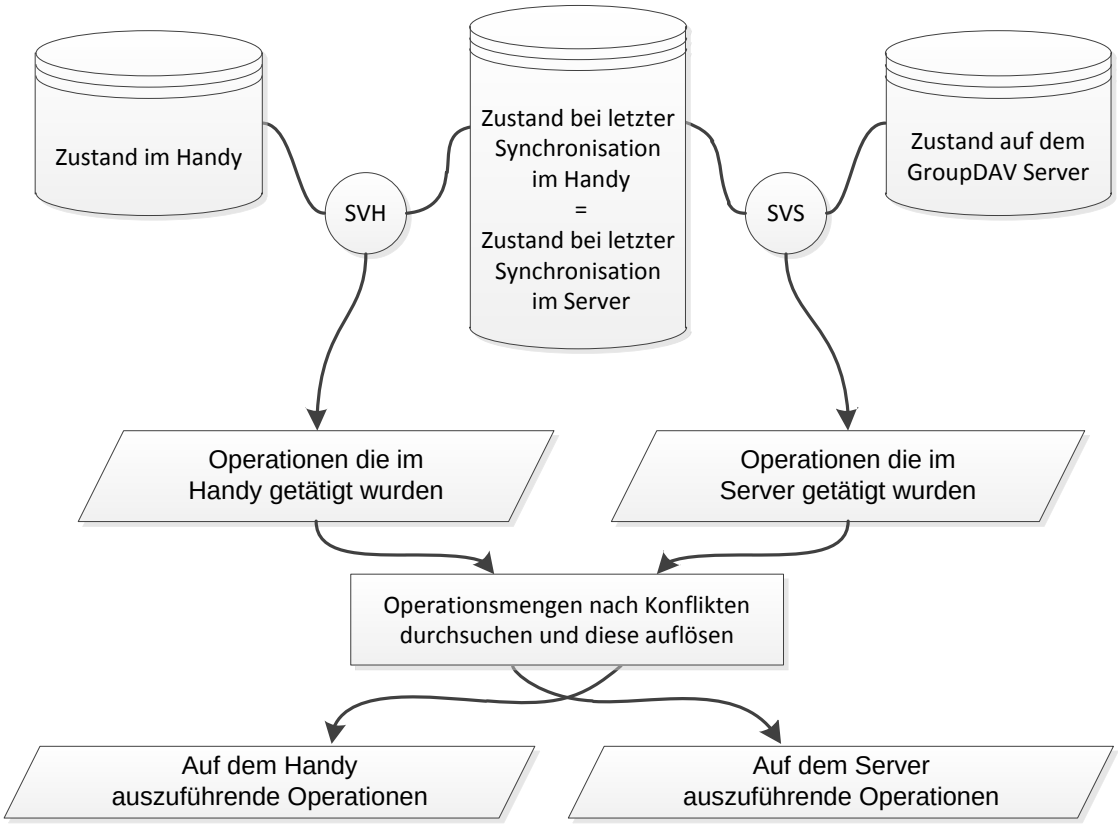


**Abbildung 3.2:** Grundlegendes Synchronisations Konzept in Gevil-Sync

| Aktueller Zustand | Sync-Zustand | Gegenseite |
|---|---|---|
| Eintrag | - | Eintrag einfügen (Insert) |
| veränderter Eintrag | Eintrag | Aktualisieren des Eintrages (Update) |
| - | Eintrag | Löschen des Eintrags (Delete) |

**Tabelle 3.1:** Erkennung von Operationen, die auf einer der beiden Seiten die synchronisiert werden, getätigt wurden.

Sync Vergleich Handy (SVH in Abbildung 3.2) ist ein Vergleich des Sync-Zustandes mit dem Handy-Zustand. Dieser findet alle Operationen, die vom User auf dem Handy ausgeführt wurden. Dazu wird zu jedem im Handy vorhandenen Eintrag die `UID` ausgelesen und im Sync-Zustand ein Eintrag mit gleicher `UID` gesucht. Das Erkennen der Operationen, die der User auf dem Server getätigt hat, funktioniert gleich und trägt den Namen Sync Vergleich Server (SVS in Abbildung 3.2).

Tabelle 3.1 zeigt die Funktionsweise von SVS und SVH. Bei Sync Vergleich Handy stellt die erste Spalte 'Aktueller Zustand' einen Eintrag auf dem Handy dar, im Falle von SVS einen Eintrag auf dem Server. Beide Operatoren werden unabhängig voneinander ausgeführt und resultieren jeweils in einer Liste von Operationen. Für den Fall, dass unter der zu einem Eintrag gehörenden UID kein Eintrag im Sync-Zustand gefunden wird, steht in Tabelle 3.1 ein -.

In der dritten Spalte steht, in welchem Fall der Vergleichs Operator ein Insert, ein Update oder ein Delete erkennt. Hat der Benutzer zum Beispiel auf seinem Handy einen neuen Kontakt erstellt, so ist dieser im Sync-Zustand noch nicht vorhanden. Dies entspricht der ersten Zeile in Tabelle 3.1, in der ein Insert erkannt wird. War ein Kontakt schon vorhanden, wurde jedoch verändert, zum Beispiel durch Einfügen einer Emailadresse, wird wie in der zweiten Tabellenspalte, ein Update erkannt. Wurde ein Kontakt gelöscht, wird wie in der dritten Tabellenzeile ein Delete erkannt.

Im Normalfall ändert der User einen Eintrag nur entweder auf dem Server oder auf dem Handy, woraufhin eine Operation auch nur auf einer Seite erkannt wird. Eine solche Operation wird die Konfliktbehandlung, siehe Abschnitt 3.3.4, unversehrt durchlaufen und dann auf der Gegenseite der Synchronisation ausgeführt. Sollte ein Eintrag auf beiden Seiten verändert worden sein, setzt die Konfliktbehandlung ein.

### 3.3.3 Effizientes Aktualisieren des Sync-Zustandes

Der Sync-Zustand ist eine Kopie des nach Synchronisationsausführung identischen Zustands im Handy und auf dem Server. Die einfachste Möglichkeit, diese Kopie aktuell zu halten ist, nach jeder Synchronisation eine Kopie vom Handy-Zustand oder vom Server Zustand anzulegen. Um die Synchronisationsausführung zeiteffizient zu gestalten, wurde hierzu jedoch eine schnellere Alternative gewählt: So lange kein Konflikt aufgetreten ist, wird jede Operation, die auf dem Handy oder auf dem Server ausgeführt wird, auch im Sync-Zustand

ausgeführt. Wurde zum Beispiel seit der letzten Synchronisation ein Kontakt vom Benutzer bearbeitet, kann dies im Sync-Zustand in nur einem Schritt widergespiegelt werden, anstatt jeden Kontakt erneut zu berühren.

Wie der Sync-Zustand im Konfliktfall aktuell gehalten wird, ist in Abschnitt 3.3.4 erläutert.

### 3.3.4 Konflikterkennung und Behandlung

Falls der Benutzer einen Eintrag auf dem Server und auf dem Handy bearbeitet, wird SVH und SVS (siehe Abbildung 3.2) eine Operation für diesen erkennen. Jede Operation führt die UID im zugehörigen VCard beziehungsweise VCalendar mit. Im Konfliktfall sind also zwei möglicherweise unterschiedliche Operationen mit gleicher UID, eine in der Ergebnismenge von SVS und eine in der Ergebnismenge von SVH, vorhanden. Die Konfliktbehandlung, welche die beiden Operationsmengen in zwei konfliktfreie Operationsmengen überführt, ist in Abbildung 3.2 durch das Viereck unten in der Mitte dargestellt.

Um eine vollständige Lösung zu erhalten, wurden alle theoretisch möglichen Konfliktsituationen in Tabelle 3.2 zusammengefasst. Das einleitende Beispiel, in dem ein Kontakt auf Handy und Server verändert wurde, wäre hier ein Update-Update Konflikt, in Tabelle 3.2 durch das Kürzel d2 auffindbar. Update-Insert, Delete-Insert und Delete-Delete Konflikte kommen jeweils doppelt in der Tabelle vor. Sie unterscheiden sich lediglich in der Reihenfolge und werden im Folgenden als identisch betrachtet.

| | Insert | Update | Delete |
|---|---|---|---|
| Insert | Insert-Insert | Update-Insert | Delete-Insert |
| Update | Update-Insert | Update-Update | Delete-Update |
| Delete | Delete-Insert | Delete-Update | Delete-Delete |

**Tabelle 3.2:** Alle möglichen Konflikte

Der häufigste Konfliktfall ist vermutlich der Update-Update Konflikt. Dieser bedeutet nicht unbedingt, dass ein Konflikt im engeren Sinne aufgetreten ist. Wenn beispielsweise auf dem Handy eine Emailadresse zu einem Kontakt hinzugefügt wurde und im Server eine andere Adresse desselben Kontaktes gelöscht wurde, tritt auch ein Update-Update Konflikt auf. Alle Konflikte dieses Types werden von der fein granularen Konfliktlösung aufgelöst. Diese zerlegt die in Konflikt stehenden Einträge in fein granulare Einträge wie Email- und Telefonnummern-Einträge, Namen, Anfangsdatum,...

Sie erkennt konzeptionell auf die gleiche Art wie die grob granulare Konfliktlösung Operationen. Es muss jedoch beachtet werden, dass fein granulare Einträge keine UID haben. Ihre Gleichheit kann nur aus der Gleichheit ihres Inhaltes geschlossen werden. Da ein veränderter fein granularer Eintrag seinem Original nicht mehr zugeordnet werden kann, können hier keine Updates erkannt werden. Stattdessen wird ein Insert des neuen Eintrages und ein Delete des Eintrages vor Veränderung festgestellt. Aus dem gleichen Grund können auch

keine fein granularen Konflikte vorkommen. Dies ist auch nicht nötig, da keiner der in der Theorie auftretenden fein granularen Konflikte eine Behandlung benötigt: Insert-Insert Konflikte führen zu einem doppelten Eintrag, solche werden aber gleich eliminiert. Delete-Delete Konflikte können nicht vorkommen, da hier auf keiner der beiden Seiten mehr ein Eintrag zur Erkennung vorhanden ist. Bei einem fein granularen Delete-Update Konflikt wird der gelöschte Eintrag entfernt und der eingefügte als Insert hinzugefügt, was semantisch dem Verwerfen des Deletes entspricht. Falls ein Feld, das nicht mehrmals vorkommen kann, wie Vorname, Nachname, Datum/Uhrzeit oder Recurrence Rule verändert wurde, wird durch Vergleich mit dem Sync-Zustand ermittelt, auf welcher Seite die Änderung stattgefunden hat und der dortige Wert übernommen. Für den Fall, dass ein Feld auf beiden Seiten verändert wurde, muss entschieden werden, welcher Wert übernommen wird. Über einen Einstellungen-Dialog ist es dem User möglich festzulegen, ob dann Wert des Servers (Server Wins), oder der Wert auf dem Smartphone (Phone Wins) verwendet wird.

Ein Delete-Delete Konflikt entsteht, wenn ein Eintrag auf beiden Seiten gelöscht wurde. Da in diesem Fall der Versuch, den betreffenden Eintrag erneut zu löschen, lediglich eine Verschwendung von Rechenzeit auslöst, werden beide Deletes verworfen und der Eintrag aus dem Sync-Zustand gelöscht.

In einigen Situationen kann es vorkommen, dass der Sync-Zustand verloren geht. Entweder durch einen Fehler oder wenn der Benutzer in den Einstellungen den Anwendungscache von Gevil-Sync löscht. In dieser Situation kommt es zu Insert-Insert Konflikten. Gevil-Sync geht nun so vor, dass die beiden zu den Insert Operationen gehörenden Einträge zuerst auf Gleichheit überprüft werden. Sind sie gleich, werden beide Inserts verworfen und der Eintrag wird im Sync-Zustand eingefügt. Für den Fall, dass sich die beiden Einträge unterscheiden, muss der Benutzer sie manuell aneinander angleichen. Solange sie ungleich sind, führt Gevil-Sync keine Änderungen an ihnen aus.

Bei den beiden Fällen Delete-Insert- und Update-Delete Konflikt, wird vorsichtshalber Daten erhaltend vorgegangen und jeweils die Delete Operation verworfen. Sollte die Löschung doch erwünscht sein, muss der Eintrag dann erneut durch den Benutzer gelöscht werden.

Beim Auftreten eines Update-Delete Konfliktes muss beachtet werden, dass der betreffende Eintrag auf einer Seite nicht vorhanden ist, sonst wäre kein Delete erkannt worden. Deshalb wird hier anstatt des Updates ein Insert ausgeführt. Das erkannte Update wird außerdem im Sync-Zustand ausgeführt, sodass auch dieser aktuell ist.

Der Update-Insert Konflikt darf bei korrekter Funktion von Gevil-Sync niemals auftreten. Ein Update kann nur vorkommen, wenn der Eintrag im Sync-Zustand vorhanden ist. Ein Insert hingegen bedeutet, dass der Eintrag im Sync-Zustand nicht vorhanden ist, was im Widerspruch zur Existenz des Updates steht. Siehe zur Verdeutlichung Tabelle 3.1. Deshalb wird Gevil-Sync in diesem Fall mit einem Fehler beendet.

#### Vorgehen bei Konflikten, deren Auflösung nicht eindeutig ist

Auch durch die fein granulare Konfliktlösung können nicht alle Konflikte gelöst werden. Bei Feldern wie Emailadresse, von denen mehrere in einem Kontakt gespeichert sein können, muss die Auflösung von Delete-Insert- und Delete-Update Konflikten erreicht werden. GevilSync geht hier Daten erhaltend vor, indem jeweils das Delete verworfen wird. Für Felder die nur einmal vorkommen können, ist dies nicht immer möglich. Zur Unterscheidung seien zwei Beispiele gegeben: Wird zu einem Kontakt auf dem Handy eine Emailadresse hinzugefügt und zur gleichen Person im Server eine Handynummer, liegt bei der Synchronisation ein Update-Update Konflikt vor. Durch die fein granulare Konfliktlösung würde dieser Konflikt automatisch so gelöst, dass die neue Emailadresse im Server und die neue Handynummer im Handy hinzugefügt wird. Im zweiten Beispiel wird der Name eines Kontaktes im Handy zu Mustermann und auf dem Server zu Musterfrau verändert. In diesem Fall ist nicht mehr nachvollziehbar, welche Änderung letztlich vom Benutzer gewünscht ist.

Da der Android Sync Manager jederzeit eine Synchronisation einleiten kann und GevilSync die Anforderung hat, den Benutzer nicht abzulenken, wird selbiger in solch einem Konfliktfall nicht nach der richtigen Operation gefragt. Im oben gegebenen Beispiel würde durch die Einstellung Phone oder Server Wins entschieden, ob als Name Mustermann oder Musterfrau übernommen wird. Zusätzlich wird im Speicher persistent ein Konflitkteintrag gespeichert, der den Benutzer über das Auftreten des Konfliktes informiert. In der GevilSync App gibt es eine Activity, die alle offenen Konflikte in einer Liste anzeigt. Bei Klick auf den Konflikteintrag wird dann ein Editor geöffnet, mit dem der betreffende Kontakt- beziehungsweise Kalendereintrag editiert werden kann. Gleichzeitig wird dadurch der Konflikt aus dem Speicher gelöscht und gilt somit nicht mehr als offen.

### 3.3.5 Operationsausführung nach Konfliktbereinigung

Erst sobald alle Operationen von Konflikten befreit sind, können sie auf der jeweiligen Gegenseite ausgeführt werden. Auf der Gegenseite deshalb, weil beispielsweise eine Operation, die vom Nutzer auf dem Handy ausgeführt wurde und deshalb auch beim Vergleich des Sync-Zustandes mit dem Handy, in Abbildung 3.3 durch SVH, erkannt wurde, auf dem Server nachgeführt werden muss. Noch konkreter: Löscht der Benutzer einen Termin auf seinem Smartphone, wird beim Vergleich des Smartphones mit dem Sync-Zustand festgestellt, dass das Event gelöscht wurde. Nun wird der dazu korrespondierende Eintrag auf dem Server, also der Gegenseite, von der Synchronisationslogik gelöscht.

Abbildung 3.3 zeigt aus einer anderen Perspektive, wie die erkannten Operationen nach Konfliktbereinigung auf Server und Smartphone ausgeführt werden und auf beiden Seiten einen identischen Zustand erzeugen.

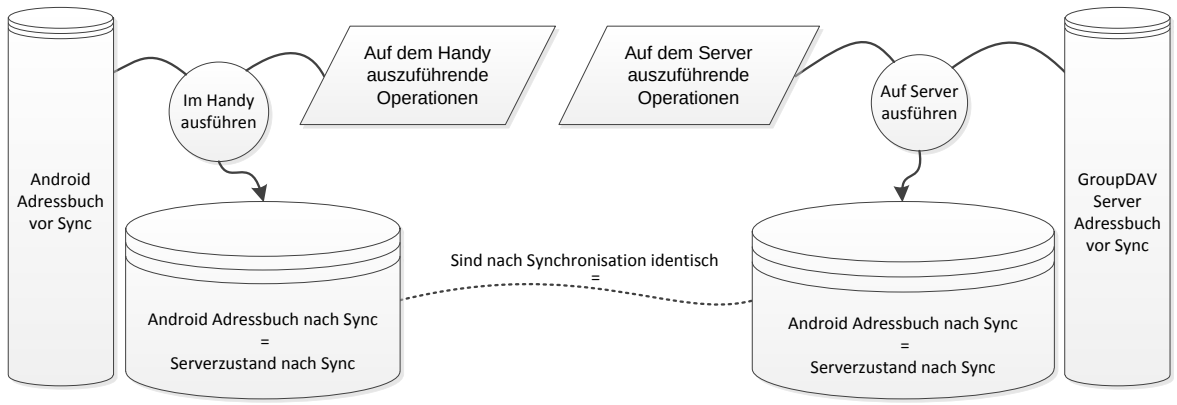


**Abbildung 3.3:** Ausführung der ermittelten Befehle auf Server und Client

### 3.3.6 Initiale Synchronisation

Abschließend soll noch der Fall betrachtet werden, dass noch nie eine Synchronisation mit Gevil-Sync stattgefunden hat. Wenn noch keine Einträge auf einer der beiden Seiten vorhanden sind, ist eine initiale Synchronisation nicht notwendig. Sobald der erste Eintrag erstellt wurde, wird dieser mit dem leeren Sync-Zustand verglichen und die Insert Operation erkannt.

Auch wenn schon Einträge auf einer oder beiden Seiten vorhanden sind, kann die entworfene Vorgehensweise beibehalten werden. Durch den zu Beginn leeren Zustand werden alle vorhandene Einträge als Insert Operation erkannt und auf die Gegenseite kopiert. Der Fall, dass eine `UID` initial auf beiden Seiten vorkommt, darf laut der Definition der `UID` eigentlich nicht vorkommen. Es könnte lediglich sein, dass Gevil-Sync, nachdem es deinstalliert wurde, wieder installiert wird. Dadurch wäre es möglich, dass ein Eintrag noch auf beiden Seiten vorhanden ist. In diesem Fall wird von der Konfliktlösung entschieden, inwiefern die beiden Einträge angeglichen werden. Existiert initial auf beiden Seiten ein Eintrag gleichen Namens, jedoch mit unterschiedlichen UIDs, so kann, falls gewünscht, unter Android der doppelte Eintrag zusammengefasst werden. Auf dem Server werden immer beide Kontakte einzeln angezeigt.

# 4 Implementierung

In diesem Kapitel wird auf Details der Implementierung von Gevil-Sync eingegangen. Dabei ist im ersten Abschnitt alles, was die Synchronisationslogik selbst betrifft, platziert. In den darauf folgenden Abschnitten wird die Implementierung von dafür nötigen Zusätzen, wie den Sync Adaptern und Einstellungen beschrieben. Das Kapitel wird von den Known Issues und Future Work abgerundet.

## 4.1 Synchronisationslogik

Im ersten Abschnitt des Implementierungskapitels geht es um die Synchronisationslogik. Darin sind alle Implementierungsdetails von Klassen und Funktionen, die bei Synchronisationsauführung genutzt werden, erläutert.

### 4.1.1 Darstellung des Sync-Zustandes im Speicher

Im Sync-Zustand ist eine Menge von Kontakten gespeichert, welche unter Verwendung von Vcards dargestellt sind. Die Daten, die in einer Vcard darstellbar sind, sind jedoch für die Synchronisation nicht ausreichend. Deshalb werden zu jedem Eintrag in der Sync-Zustand Liste, die Android `_id`, der Dateiname auf dem Server, sowie der Etag Wert beim letzten Abgleich mit dem Server als Metadaten gespeichert.

Die Android `_id` wird benötigt, um auf einen Kontakt im Android Adressbuch zuzugreifen. Für einen Kontakt, der gerade vom Server geladen, aber noch nicht im Adressbuch des Handys eingefügt wurde, ist es möglich, dass dieser `_id` -Wert noch nicht gesetzt ist. Erst nach dem Einfügen im Android Adressbuch existiert sie und wird im Sync-Zustand gespeichert.

GroupDAV verwendet zur Identifikation von VCards die `UID`. Dieses Vorgehen wurde in Gevil-Sync fortgeführt. Um auf eine auf dem Server befindliche VCard zuzugreifen, ist zusätzlich zur UID noch ihr dortiger Dateiname notwendig. Dieser kann identisch zur UID der VCard sein, muss es aber nicht. Auch der Etag (Abschnitt 2.1.2), welcher in CalDAV und dem hier eingesetzten GroupDAV Protokoll gleichermaßen Verwendung findet, muss zu jedem Eintrag bekannt sein.

Realisiert wurde die Speicherung dieser Metadaten in der Klasse `GevilVCard`, in der auch der Zugriff auf die Einträge in der VCard zentralisiert wurde. Außerdem implementiert sie die Methode `Comparable<GevilVCard>`, mit der entschieden wird, ob zwei `GevilVCards`

im Sinne der Synchronisation identisch sind. Das heißt, wenn sie ungleich sind, muss ein Update durchgeführt werden.

Die Klasse `AktuellGevilVCard` erweitert die `GevilVCard`. Sie stellt eine VCard dar, die nicht aktualisiert werden muss und wird verwendet, wenn schon durch den Etag Vergleich ermittelt wurde, dass keine Änderung der VCard stattgefunden hat. Beim Vergleich einer `AktuellGevilVCard` mit jeder `GevilVCard` wird die Gleichheit festgestellt.

Bei der Darstellung von Kalendereinträgen wird keine VCard, dafür jedoch ein VCalendar verwendet. Da der Zugriff auf VCalendar anders als bei VCards funktioniert, wurde die Klasse `GevilVCalendar` eingeführt. Neben dem Zugriff auf Properties und Parameter sind in der `GevilVCalendar` Klasse die gleichen Metadaten wie in der `GevilVCard` Klasse enthalten.

### 4.1.2 Erweiterung des VCard Funktionsumfanges durch GevilVCardHash

Das Suchen von Einträgen verschiedener Typen wird von ical4j-vcard nicht angeboten. Deshalb wurde der Funktionsumfang der Bibliothek durch die Klasse `GevilVCardHash` erweitert. Diese Klasse verwendet zwei HashMaps. In der ByValue Hashmap wird am Beispiel Emailadresse, ihr Adresseintrag und Adresstyp durch den Adressstring gefunden. Die ByType Hashmap gibt für einen Typ wie zum Beispiel `EMAIL_HOME` einen Emaileintrag zurück. Es ist auch möglich, dass für einen Typ mehrere Adressen in einer VCard gefunden werden können. Der `GevilVCardHash` Klasse wird im Konstruktor übergeben, welche Art von Properties auf der VCard gesucht werden sollen. Verwendete Arten von Properties waren Emailadressen und Telefonnummern. Unter anderem beim Einfügen ins Adressbuch ist eine notwendige Funktion, die von `GevilVCardHash` bereitgestellt wird, Einträge nur einmal zu finden. Dies wird realisiert, indem gefundene Einträge mit einem Flag markiert werden. Auch für die `compareTo()`-Methode der `GevilVCard`, welche `Comparable<GevilVCard>` implementiert, ist die `GevilVCardHash` Klasse obligatorisch, weil durch sie herausgefunden wird, ob eine Property einer VCard A auch in der Liste von Properties der Vergleichs VCard B vorhanden ist. Durch das gesetzte Gefunden-Flag kann abschließend herausgefunden werden, ob jede Property der VCard B auch in A vorhanden war.

### 4.1.3 Effizientes Auslesen von Kontakten aus dem Android Adressbuch

In den Grundlagen wurde bereits erläutert, wie prinzipiell auf das Android Adressbuch zugegriffen werden kann. Dabei wurden jedoch alle Kontakte aus dem Adressbuch ausgelesen und zur Erkennung von Gevil-Sync Kontakten müsste jeder Kontakt erneut auf seine Accountzugehörigkeit überprüft werden. Dieses Vorgehen war auf dem Testgerät, auf dem sich viele nicht Gevil-Sync Kontakte befanden, relativ langsam. Deshalb wurde der Kontakt Auslesevorgang in zwei Datenbankabfragen unterteilt: In der Ersten werden unter Verwendung der unten stehenden, SQL ähnlichen, Abfrage alle `RawContactId`'s von Adressbuchkontakten, die zu Gevil-Sync gehören, abgefragt. Die Abfrage:

```
SELECT account_type, account_name, contact_id, _id
FROM ContactsContract.RawContacts.CONTENT_URI
WHERE RawContacts.ACCOUNT_TYPE = Constants.ACCOUNT_TYPE
```

wurde für die Verwendung in Android so ausgedrückt:

```
String[] projection = new String[] {"account_type", "account_name", "contact_id", "_id"};
cur = resolver.query(ContactsContract.RawContacts.CONTENT_URI, projection,
    ContactsContract.RawContacts.ACCOUNT_TYPE + "=?", new String[]{Constants.ACCOUNT_TYPE},
    null);
```

Android bietet die Möglichkeit, wie in Abbildung 4.1, ähnliche Kontakte zu einem Kontakt zusammenzufassen. In der Grafik wurden zwei Gevil-Sync Kontakte mit einem Facebook Kontakt und einem Kontakt von der Simkarte des Handys zusammengefasst. Der zusammengefasste Kontakt im Adressbuch zeigt alle vorhandenen Informationen auf einer Seite an. Durch Verwendung des Profilbildes von Facebook kann der Kontakt beispielsweise optisch aufgewertet werden. Aufgrund solcher zusammengefasster Kontakte gibt es zwei oft unterschiedliche Ids für jeden Kontakt, nämlich die `RawContactId` und die `ContactId` (im Testgerät mit dem Spaltenname `_id`). Über die `ContactId` kann auf den zusammengefassten Kontakt zugegriffen werden, sie wäre in Abbildung 4.1 für alle Kontakte gleich. Jeder der vier Raw-, das heißt nicht zusammengefassten Kontakte, hat eine eigene `RawContactId`, die ihn eindeutig identifiziert.

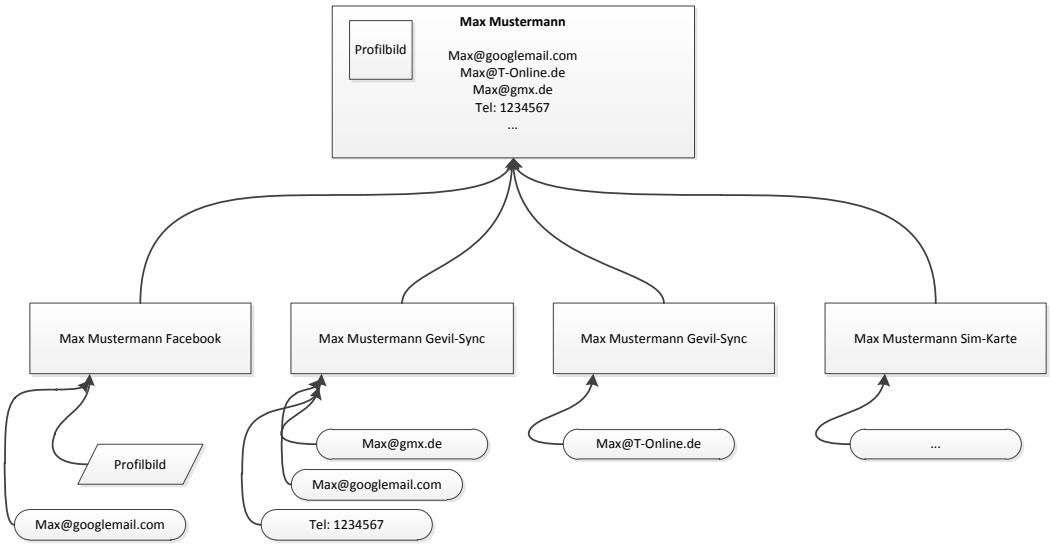


**Abbildung 4.1:** Zusammengefasster Kontakt in Android

Gevil-Sync würde weiterhin beide Gevil-Sync-Kontakte einzeln mit dem Server synchronisieren und die anderen Raw-Kontakte ignorieren. Die für uns relevanten Kontaktinformationen

finden sich unter der Tabelle Data.CONTENT_URI. Ein direkter Zugriff durch die RawContactId ist hier nicht möglich, es kann nur der komplette zusammengesetzte Kontakt via ContacId ausgelesen werden. Unter Verwendung der eben beschriebenen Abfrage haben wir eine Liste aller contactId's und rawContactId's von Gevil-Sync Kontakten erhalten. An Hand dieser rawContactId's werden wir gleich entscheiden, ob wir einen Raw-Kontakt ignorieren. Durch das WHERE Statement haben wir Ids zusammengesetzter Kontakte, in denen kein Raw-Kontakt zu Gevil-Sync gehört, ausgeschlossen. Die folgende Query liest, unter Verwendung eines Joins über die contact_id, alle Raw-Kontakte, die unter dieser contact_id zusammengefasst sind, aus:

```
String[] PROJECTION = new String[]{Data._ID, Data.MIMETYPE, Data.DATA1, Data.DATA2,
    Data.DATA3, Data.RAW_CONTACT_ID, Data.CONTACT_ID, };

Cursor c = resolver.query(Data.CONTENT_URI, PROJECTION, Data.CONTACT_ID + "=?", new String[]
    {String.valueOf(contact_id)}, null);
```

Wie aus dem SELECT Statement zu erkennen ist, gibt die zweite Query einen Cursor auf eine Tabelle mit 7 Spalten zurück. In der Tabellenspalte Data.MIMETYPE steht, um welche Art von fein granularem Eintrag, also zum Beispiel Structured Name, Emailadresse oder Telefonnummer, es sich bei der Tabellenzeile handelt. Im Falle einer Emailadresse findet sich unter Data.Data1 die Adresse selbst, unter Data.Data2 ist der Typ festgehalten, möglich wären hier Werte wie Email.Home oder Email.Work. Die Spalte Data.Data3 kann für weitere, spezifische Informationen verwendet werden.

Ist ein Gevil-Sync-Kontakt von Android mit einem nicht Gevil-Sync-Kontakt verknüpft worden, wird auch der nicht Gevil-Sync-Kontakt von der Query ausgelesen. Da uns aus der ersten Query bekannt ist, welche rawContactId's zu Gevil-Sync gehören, ist das Ziel nun, nur diese auszulesen und alle anderen zu überspringen. Beachtet werden muss dabei auch, dass durchaus mehr als ein Gevil-Sync-Kontakt in einem zusammengesetzten Kontakt vorkommen könnte. Gelöst wurde dies durch Verwendung einer HashMap, welche als Key die RawContactId und als Value eine GevilVCard enthält. Eine Methode namens readContactByContactId() liest alle Raw-Kontakte, die unter der gegebenen ContactId zusammengefasst sind, in diese HashMap ein. Dazu liest sie aus jeder Zeile zuerst die RawContactId und dann den fein granularen Eintrag. Abgelegt wird jeder fein granulare Eintrag dann unter der GevilVCard, die die HashMap unter dem Schlüssel RawContactId beinhaltet. Die Methode readContactByRawContactId() gibt nach Aufruf von readContactByContactId() den, in der HasMap unter der gewünschten RawContactId gespeicherten, Kontakt zurück.

### 4.1.4 Schreiben ins Android Adressbuch

Wie bereits erwähnt, speichert Android alle Adressen in einem Content Provider, welcher auf einer SQLite Datenbank basiert. Schreibzugriffe können also via Insert, Update und Delete stattfinden.

### Einfügen eines Kontaktes ins Android Adressbuch

Als erstes soll gezeigt werden, wie ein neuer Kontakt ins Android Adressbuch eingefügt wird. Nach dem Kontaktnamen wird ein Eintrag, der anzeigt dass es sich um einen Gevil-Sync Account handelt geschrieben.

```
Uri uri =
    Data.CONTENT_URI.buildUpon().appendQueryParameter(ContactsContract.CALLER_IS_SYNCADAPTER,
    "true").build();
ops.add(ContentProviderOperation.newInsert(uri)
       .withValue(ContactsContract.RawContacts.ACCOUNT_NAME, AccountName)
       .withValue(ContactsContract.RawContacts.ACCOUNT_TYPE, Constants.ACCOUNT_TYPE)
       .build());
```

Zum einfügen jedes Emailadresseintrages wird die folgende Routine verwendet. Da ein neuer Kontakt eingefügt wird, ist `newContact=true`. Beim Update eines Kontaktes wird die gleiche Routine mit `newContact=false` verwendet. Der Integer `mBackReference` entspricht der Länge der ArrayList `Ops`, vor der neue Kontakt eingefügt wurde [devb]. Auch hier funktioniert das Einfügen einer Telefonnummer gleich. Abschließend wird noch die von Gevil-Sync verwendete UID in ein zusätzliches Feld eingetragen.

```
ContentProviderOperation.Builder builder = ContentProviderOperation.newInsert(uri);

if(newContact){
       builder.withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID, mBackReference);
}else{
       builder.withValue(ContactsContract.Data.RAW_CONTACT_ID, RawContactId);
}

builder.withValue(ContactsContract.Data.MIMETYPE, Constants.MIMETYPE_UID);
builder.withValue(ContactsContract.Data.DATA1, username);
builder.withValue(ContactsContract.Data.DATA2, "Gevil-Sync Profil");
builder.withValue(ContactsContract.Data.DATA3, uid);
```

### Löschen eines Kontaktes aus dem Adressbuch

Auch ein zu löschender Kontakt wird mittels der `rawContactId` identifiziert. Anders beim Löschvorgang ist, dass diese mit dem Befehl `withAppendedId` an die `RawContacts.CONTENT_URI` angefügt wird. Durch den `CALLER_IS_SYNCADAPTER` Parameter wird Android signalisiert, dass der Zugriff durch einen Sync Adapter geschieht. `Ops` ist eine ArrayList mit `ContentProviderOperations`. Erst sobald sich ca. 50 Operationen angesammelt haben, werden diese ausgeführt. Auf diese Art wird der Zugriff beschleunigt.

```
Uri uri = ContentUris.withAppendedId(RawContacts.CONTENT_URI, rawContactId);
uri.buildUpon().appendQueryParameter(ContactsContract.CALLER_IS_SYNCADAPTER, "true").build();
builder = ContentProviderOperation.newDelete(uri);
ops.add(builder.build());
```

#### Update eines Kontaktes im Android Adressbuch

Wurde ein Kontakt auf dem Server vom User verändert, muss die Update `GevilVCard`, welche die aktualisierten Daten beinhaltet, über den veralteten Kontakteintrag im Android Adressbuch geschrieben werden. Dazu muss der Kontakt erst aus dem Android Adressbuch ausgelesen werden. Identifiziert wird er durch die `rawContactId`. Dabei entspricht wieder jede Zeile in der Datenbank einem fein granularen Eintrag, beziehungsweise einer Property in der VCard. Um welche Art von fein granularem Eintrag es sich handelt, ist auch hier in der MIMETYPE Spalte gespeichert. Für jeden fein granularen Eintrag wird nun durch Vergleich mit der Update `GevilVCard` entschieden, ob er nicht verändert, upgedatet oder gelöscht werden muss. Auch ist es möglich, dass ein neuer fein granularer Eintrag per Insert ins Android Adressbuch eingetragen werden muss. Nachdem per Query, wie in Abschnitt 4.1.3, der Cursor c, der die einzelnen fein granularen Einträge des Kontaktes beinhaltet, erstellt wurde, wird mit seiner Hilfe die `Uri`, die `id` sowie der Inhalt der MIMETYPE Spalte, welche angibt, von welchem Type der fein granulare Eintrag ist, ausgelesen:

```
while (c.moveToNext()) {
       final long id = c.getLong(DataQuery.COLUMN_ID);
       uri = ContentUris.withAppendedId(Data.CONTENT_URI, id);
       final String mimeType = c.getString(DataQueryContactId.COLUMN_MIMETYPE);
}
```

Handelt es sich bei der Datenbankspalte zum Beispiel um einen Emailadresseintrag:
`if (mimeType.equals(CommonDataKinds.Email.CONTENT_ITEM_TYPE))`

so wird in der `GevilVCard` nach einem Eintrag gleicher Adresse oder Types gesucht und dieser, falls er sich geändert hat, per Update ins Adressbuch eingetragen. Identifiziert wird der zu aktualisierende fein granulare Eintrag an Hand der `uri`, das Vorgehen ist bei anderen Eintragstypen (vor allem Telefonnummern) dasselbe.

```
uri = uri.buildUpon().appendQueryParameter(ContactsContract.CALLER_IS_SYNCADAPTER,
    "true").build();

ContentProviderOperation.Builder builder;
builder = ContentProviderOperation.newUpdate(uri);
builder.withValue(ContactsContract.CommonDataKinds.Email.DATA, emailAdresse);
builder.withValue(ContactsContract.CommonDataKinds.Email.TYPE, androidEmailType);
ops.add(builder.build());
```

Als nächstes betrachten wir das Einfügen einer neuen Emailadresse zu einem bestehenden Kontakt. Nachdem alle Einträge des Kontaktes aus dem Adressbuch ausgelesen und mit der `GevilVCard` abgeglichen wurden, wird festgestellt, dass es immernoch eine Adresse in der `GevilVCard` gibt, die nicht eingetragen ist. Hier kommt die gleiche Routine wie beim Einfügen von fein granularen Einträgen, im Zuge des Inserts eines neuen Kontaktes, zum Einsatz. An dieser Stelle ist jedoch `newContact=false`.

```
Uri uri =
    Data.CONTENT_URI.buildUpon().appendQueryParameter(ContactsContract.CALLER_IS_SYNCADAPTER,
    "true").build();
ContentProviderOperation.Builder builder = ContentProviderOperation.newInsert(uri);
```

```
builder.withValue(ContactsContract.Data.MIMETYPE,
    ContactsContract.CommonDataKinds.Email.CONTENT_ITEM_TYPE);
builder.withValue(ContactsContract.CommonDataKinds.Email.DATA, emailAdresse);
builder.withValue(ContactsContract.CommonDataKinds.Email.TYPE, androidType);

if(newContact){
       builder.withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID, mBackReference);
}else{
       builder.withValue(ContactsContract.Data.RAW_CONTACT_ID, RawContactId);
}

ops.add(builder.build());
```

Die neue Zeile wird hier einfach in die Datenbank unter `Data.CONTENT_URI` eingetragen. In diesem Fall handelt es sich um einen bereits existierenden Kontakt, deshalb ist `newContact=false`. Durch die `RawContactId` wird die Zugehörigkeit des Eintrages zu einem Kontakt sichergestellt.

Schließlich kann es auch vorkommen, dass die Update `GevilVCard` weniger Einträge als der Android Kontakt enthält, woraufhin ein fein granulares Delete durchgeführt wird. Nachdem die `Uri` einer Zeile ausgelesen wurde und kein Wert zum Aktualisieren in der `GevilVCard` gefunden werden konnte, wird die entsprechende Zeile gelöscht.

```
builder = ContentProviderOperation.newDelete(uri);
ops.add(builder.build());
```

### 4.1.5 Speicherung der UID im Adressbuch und Kalender

Unter Android werden Kontakte über die `_id` identifiziert. Es ist jedoch nicht sichergestellt dass, nachdem ein Kontakt gelöscht wurde, ein neu erstellter nicht wieder dieselbe `_id` wie der gelöschte hat. Dasselbe gilt auch für Termine im Android Kalender, welche dort ebenfalls per `_id` identifiziert werden. Aus diesem Grund wurde vermieden, ein Mapping von der Android `_id` auf die UID zu verwenden. Stattdessen speichert Gevil-Sync die UID zu jedem Kontakt- und Kalendereintrag unter einem extra Feld im jeweiligen Content Provider. Das Mapping von der UID zur `_id` wird implizit dadurch hergestellt, dass beide in den Klassen `GevilVCard` und `GevilVCalendar` abgelegt sind.

Zum Speichern eines zusätzlichen Wertes im Adressbuch, ist ein `<meta-data>` Eintrag in der Manifest Datei notwendig, welcher auf eine xml Datei, in Gevil-Sync die contacts.xml, verweist. Außerdem muss die Beschriftung der Spalte, welche die zusätzlichen Daten beinhaltet, festgelegt werden. Gevil-Sync verwendet die Bezeichnung "vnd.android.cursor.item/com.gevil.profile". Im Falle von Kalendereinträgen konnte der zusätzliche Wert einfach unter der Spalten `sync_source` abgelegt werden.

Hat der Benutzer unter Verwendung der Android Kalender- oder Adressbuch App, einen neuen Eintrag erstellt, so ist dessen UID Feld noch leer. Erst sobald Gevil-Sync diesen das erste Mal liest, wird eine UID generiert und sofort in den betreffenden Content Provider zurück geschrieben. Um sicherzustellen, dass die UID wirklich einmalig ist, findet bei der

Generierung eine Kombination aus `_id`, einem Geräte Identifier, einer Zufallszahl, sowie dem aktuellen Datum, Einsatz.

### 4.1.6 Erkennung von getätigten Operationen

In diesem Abschnitt wird die Implementierung der beiden Sync Vergleich Methoden erläutert. Bei SVH und SVS wird gleich (Algorithmus 4.1) vorgegangen. Für SVH wird der `Aktuelle-Zustand` durch das Handy Adressbuch beziehungsweise den Kalender repräsentiert, für SVS durch dieselben Einträge auf dem Server. Nach Ablauf von Algorithmus 4.1 ist das Ergenis in der `Operationsmenge` gespeichert. Diese Ergebnismenge ist, in Abbildung 3.2 des Entwurfskapitels, im Falle von SVH durch „Operationen die im Handy getätigt wurden“ und nach Ausführung von SVS durch „Operationen die im Server getätigt wurden“, dargestellt.

**Algorithmus 4.1** Erkennung getätigter Operationen, SVS und SVH

```
Menge von Operationen: Operationsliste
Menge von Einträgen: Sync-Zustand
Menge von Einträgen: Aktueller-Zustand = Server- oder Handyzustand
Eintrag s, z

for all s ∈ Aktueller-Zustand do
    z := Suche den Eintrag, der die gleiche UID wie s hat, im Sync-Zustand
    if kein z gefunden then
        Operationsliste.add(Insert s)
    else
        z markieren
        if z ≠ s then
            Operationsliste.add(Update z -> s)
        end if
    end if
end for

for all z ∈ Sync-Zustand do
    if e nicht markiert then
        Operationsliste.add(Delete z)
    end if
end for
```

Die erste for-Schleife aus Algorithmus 4.1 iteriert über alle Einträge aus dem Handy (Server) und sucht den korrespondierenden Eintrag im Sync-Zustand. Durch Verwendung einer HashMap findet die Suche in konstanter Zeit statt. Falls dabei nichts gefunden wird, wird ein Insert erkannt (Begründung, siehe Tabelle 3.1). Wird ein korrespondierender Eintrag (`z`) gefunden, findet eine Markierung selbigens im Sync-Zustand statt und er wird mit dem Eintrag (`s`) auf dem Handy (Server) verglichen. Sind die beiden gleich, gilt der Eintrag als

unverändert und es muss nichts getan werden. Andernfalls wird eine Update Operation hinzugefügt.

Nun stellt sich noch die Frage, wie erkannt wird, falls der Benutzer einen Eintrag von seinem Handy (Server) gelöscht hat. Der gelöschte Eintrag war nach Abschluss der letzten Synchronisation noch auf beiden Seiten der Synchronisation vorhanden und ist deshalb auch noch im Sync-Zustand vertreten. Da er im Handy (Server) gelöscht wurde, hat die erste For Schleife in Algorithmus 4.1 ihn nicht berührt. Deshalb iteriert die zweite Schleife über jeden Eintrag im Sync-Zustand und fügt, falls dieser nicht schon in der ersten Schleife markiert wurde, eine Delete Operation in die Liste der, auf der Gegenseite auszuführenden, Operationen `Operationsliste` ein.

### 4.1.7 Konflikterkennung

Vor nun alle erkannten Operationen auf Server, beziehungsweise Client, ausgeführt werden, müssen sie noch nach Konflikten durchsucht werden. Hierbei kommt erneut die UID zum Einsatz. Da zu jeder Operation eine `GevilVCard` oder ein `GevilVCalendar` gehört, ist die dazugehörige UID vorhanden. Ein Konflikt liegt vor, wenn auf Handy und Server eine Operation für den selben Eintrag gefunden wurde. Dazu werden die beiden Operationsmengen, welche Ergebnis von SVS und SVH sind, herangezogen. Die Konflikterkennung baut eine HashMap über eine der beiden auf, welche die UID als Key und die voraussichtlich auszuführende Operation als Value beinhaltet. An Hand dieser werden nun alle Operationen, die in beiden Mengen vorhanden sind, gefunden.

Die im Entwurf konzipierte fein granulare Konflikterkennung für Kontakte ist durch 3 Listen mit Properties implementiert: Eine für die VCard des Kontaktes auf dem Handy, eine für die des Sync-Zustandes und eine für die des Servers. In jeder Liste werden alle Properties sortiert eingefügt. Existiert eine Property, weil sie zum Beispiel gelöscht wurde, in einem der drei Zustände nicht, so wird an dieser Stelle eine leere Zeile in die Tabelle eingefügt. Durch Nebeneinanderstellen der drei Tabellen, können nun die zwei möglichen fein granularen Operationen, Insert und Delete, erkannt werden.

Properties die mehrfach vorkommen können, gibt es nur in VCards. Andere Properties, wie zum Beispiel die Location im VCalendar oder der Vorname in der VCard, können nur einmal je Eintrag vorkommen. In diesen Fällen wird durch einen Stringvergleich mit dem entsprechenden Feld im Sync-Zustand herausgefunden, auf welcher Seite die Änderung stattgefunden hat und dementsprechend der aktuelle Wert übernommen.

### 4.1.8 Ausführung von Operationen

Erkannte Operationen müssen unter Android und auf dem Server ausgeführt werden. Da sich die Operationstypen am beim Zugriff auf die Android Content Provider verwendeten SQLite Jargon orientieren, ist ihre Ausführung im verantwortlichen ContentProvider naheliegend: ein Update wird als Update ausgeführt, ein Insert als Insert und ein Delete als Delete.

Neben der Android nahen Darstellung müssen die drei Operationstypen auch für die Ausführung auf dem Server umgesetzt werden. Im Falle des Inserts wird die VCard oder der VCalendar via Http-Put hochgeladen. Auch das Update geschieht mittels eines Http-Put Befehls. Dadurch, dass die UID die gleiche ist, wird der Eintrag auf dem Server einfach überschrieben. Das Löschen eines Eintrages auf dem Server geschieht mit dem Dateinamen, welcher in der jeweiligen `GevilVCard` oder `GevilVCalendar` gespeichert ist.

### 4.1.9 Unterschiede bei der Implementierung für Kalendereinträge

Der Entwurf von Gevil-Sync ist für Kalendereinträge und Adressbuchkontakte derselbe. Der auffälligste Unterschied ist wahrscheinlich, dass anstatt der VCard ein VCalendar Einsatz findet. Das Lesen und Schreiben von Android Kalendereinträgen unterscheidet sich gänzlich vom Zugriff auf das Adressbuch. Das Vorgehen ist im Grundlagenkapitel unter Abschnitt 2.3.1 erläutert. Das `ContentValues` Objekt, welches in den Content Provider eingefügt wird, wird von der Klasse `GevilVCalendar` in der Methode `getContentValues(...)` erzeugt.

Gevil-Sync erstellt einen eigenen Kalender für jeden, im entsprechenden Android Menu angelegten, Gevil-Sync Account und nimmt nur an diesem Veränderungen vor. Dazu wurde die Methode `findOrCreateCalendar(String UserName)` implementiert. Diese sucht in der Content Provider Tabelle unter `content://com.android.calendar/calendars`, in der alle Kalender aufgelistet sind, einen Gevil-Sync Kalender mit dem gewünschten Nutzernamen und gibt dessen `calendar_id` zurück. Falls kein passender Eintrag gefunden wird, erstellt die Methode einen entsprechend neuen Gevil-Sync Kalender. Dabei müssen die drei Einträge:

```
cv.put("access_level", 700);
cv.put("sync_events", 1);
cv.put("timezone", "Europe/Amsterdam");
```

hinzugefügt werden. `cv` ist hier das `ContentValue` Objekt. Der `access_level 700` Eintrag erlaubt, dass Termine im Android Editor verändert werden. Ohne die anderen beiden Einträge weigert sich der Android Kalender, den Termin anzuzeigen [deva] [Cal].

Auch bei der Kommunikation mit dem GroupDAV Server gab es einen kleinen Unterschied: Hier musste beim HTTP-Put der Entity Type von `text/vcard` nach `text/calendar` geändert werden, damit selbiger das Überschreiben existierender Einträge ermöglichte.

### 4.1.10 Mapping zwischen Android- und Vcalendar- Zeitzonen

Im Android Kalender wird ein Datum inklusive Tag und Uhrzeit als einziger Long Wert in Millisekunden abgespeichert. Im VCalendar Dateiformat wird eine solche Zeit im Format JJJJ/MM/TT/'T'/HH/MM/SS dargestellt. Dabei steht J für Jahr, M für Monat, T für Tag, 'T' für den Character T, H für die Stunde, M für Minute und S für Sekunde.

Durch ein `java.util.Calendar` Objekt (hierbei handelt es sich nicht um einen VCalendar), kann diese Millisekunden-Zeitangabe in Jahr, Monat, Tag,... umgerechnet werden. Das

VCalendar Zeitformat wird daraus dann mit folgender Codezeile gewonnen: `String ergebnis = jahr + monat + tag + "T"+ stunde + minute + sekunde;`. Dieses `Ergebnis` kann nun als `DtStart` oder `DtEnd` in einen VCalendar eingetragen werden.

Auch die Umwandlung des VCalendar-Formats in die, von Android verwendete, Millisekunden Darstellung ist für die Synchronisation notwendig. Dabei findet die Klasse `Date` Einsatz:

```
if(timeProp.endsWith("z")){
       d = new Date(jahr-1900, monat-1, tag, stunde, minute , sekunde);
}else{
       d = new Date(jahr-1900, monat-1, tag, stunde, minute + zeitZonenOffsetInMinuten ,
           sekunde);
}
```

Wenn am Ende des Datum-Wertes ein z steht, ist die Zeitangabe im Universal Time Zone Identifier (UTC) -Format. Will man beispielsweise wissen, wie spät es zu einer gegebenen UTC Uhrzeit im mitteleuropäischen Zeitbereicht ist, muss man eine Stunde addieren. Falls der Zeitstring nicht mit z endet, muss ein `TzId`-Wert im `VCalendar` angegeben sein. Für `TZID=Europe` ist dann `zeitZonenOffsetInMinuten=60`; für London ist der Offset 0 [dat].

### 4.1.11 Mapping zwischen Android- und Vcalendar- Wiederholungsregeln

Im Android Adressbuch sowie im VCalendar Dateiformat werden sogenannte Recurrence-Rules, zu deutsch Wiederholungsregeln verwendet, um sich wiederholende Termine darzustellen [FD98a]. Anwendungsbeispiel für eine solche Wiederholungsregel sei ein Geburtstag. Dieser findet jedes Jahr statt und es gibt unendlich viele Wiederholungen. In der Praxis gibt es allerdings doch einen Unterschied, sodass die Wiederholungsregel von Android und dem GroupDAV Server nicht unverändert übernommen werden konnten. Android fügt zusätzlich den Parameter `WKST=MO` in den Regel-String ein. Beim Auslesen aus Android wird dieser Wert, der den ersten Tag der Woche angibt, einfach entfernt. Andersherum, beim Einfügen eines `GevilVCalendars`, wird er zum Wiederholungsregel-String hinzugefügt.

Beim Einfügen eines Termins mit Wiederholungsregel in den Kalender, müssen in dessen Datenbankzeilen zusätzlich die Zeilen

```
cv.put("visibility", 0);
cv.put("duration", dauer);
```

eingetragen werden. Der eigentliche Termin-Eintrag muss dann auf unsichtbar gesetzt werden, da nur seine Wiederholungen im Kalender angezeigt werden. Der ansonsten obligatorische `DtEnd`-Eintrag darf in einem sich wiederholenden Termin nicht vorkommen. Stattdessen muss hier ein `duration`-Eintrag beigefügt werden. Dieser gibt schlicht die Zeitdifferenz zwischen `DtStart` und `DtEnd` an, wird jedoch in einem völlig anderen Format dargestellt [deva].

### 4.1.12 Synchronisation von Remindern

Gevil-Sync synchronisiert auch Terminerinnerungen. Im VCalendar werden diese nicht in der `VEVENT Component`, sondern in einer `VALARM Component` gespeichert. Im `VALARM` wird der Reminder unter der Property TRIGGER abgespeichert.

Unter Android kommt zur Speicherung der Reminder erneut eine eigene Tabelle zum Einsatz. Dieser war im Testgerät unter der URI `content://com.android.calendar/reminders` auffindbar. Gespeichert wird dort schlicht ein Minuten Integer. Dieser gibt an, wie viel Minuten vor Beginn des Events der Alarm ausgelöst werden soll. Die Zugehörigkeit zu einem Kalendereintrag wird per Join über das `_id` Feld des Events und das `event_id` Feld in der Reminder Tabelle hergestellt. Durch das Feld `_has_reminder`, welches für jeden Kalendereintrag gespeichert ist, kann festgestellt werden, ob in der Reminder Tabelle ein entsprechender Eintrag vorhanden ist.

## 4.2 Gevil-Sync Sync Adapter

Um einen Gevil-Sync Account mit zugehörigem Sync Adapter unter Android einzurichten, muss der Benutzer seinen Benutzernamen und Passwort eingeben. Danach überprüft Gevil-Sync die Richtigkeit der Eingabe beim Server. Dazu hängt es die Anmeldeinformationen an ein PROPFIND an, welches es zur Serveradresse schickt. Bei korrekten Anmeldedaten hat die Antwort darauf den Statuscode Multi Status. Andere HTTP Aufrufe können an dieser Stelle nicht verwendet werden, weil der Statuscode der Antwort hierauf immer ein OK ist. Bei falschem Passwort antwortet der eingesetzte Server mit einer HTML Seite, die Teil eines Browser Interfaces ist, welches zur Eingabe der korrekten Anmeldedaten auffordert. Diese HTML Seite hat auch den Statuscode OK.

Nach erfolgreicher Einrichtung eines Gevil-Sync Accounts unter Android ist darin der Nutzername und das Passwort zur Serverkommunikation gespeichert. Daduch muss der User seine Anmeldeinformationen nur einmal eingeben. Das Auslesen eines oder mehrerer Gevil-Sync Accounts erfolgt so:

```
mAccountManager = AccountManager.get(context);
Account[] ac = mAccountManager.getAccountsByType(Constants.AUTHTOKEN_TYPE);

for(int i = 0; i< ac.length; i++){
        String pwd = "passwort not found.";
        pwd = mAccountManager.blockingGetAuthToken(ac[i], Constants.AUTHTOKEN_TYPE, false);
        // weitere Aktionen mit den gelesenen Account Informationen
}
```

Im Account Array `ac` ist ein Verweis auf alle Gevil-Sync Kontakte gespeichert, wobei die Account Zugehörigkeit durch den String `Constants.AUTHTOKEN_TYPE` sichergestellt wird. Nun wird jeder einzelne Account durchiteriert. Auf den Namen des Accounts kann per

`ac[i].name` zugegriffen werden, das Auslesen des Passworts ist im Beispielcode illustriert. Infolgedessen können im Schleifenrumpf weitere Aktionen mit den Anmeldedaten ausgeführt werden.

Wenn ein Konto samt zugehörigem Sync Adapter gelöscht wird, löscht Android automatisch alle zu diesem Konto gehörigen Kalenderevents und Adressbucheinträge, nicht jedoch den Sync-Zustand von Gevil-Sync. Falls dann nach dem Löschen des Sync Adapters selbiger wieder hinzugefügt wird und die Synchronisation ausgeführt, sind alle Einträge auf dem Handy gelöscht, was durch die Synchronisation erkannt wird. Das Resultat ist, dass fälschlicherweise alle Einträge gelöscht werden. Um dieses Fehlverhalten zu vermeiden, löscht Gevil-Sync in dieser Situation auch den Sync-Zustand. Außerdem muss bei Ausführung einer Synchronisation explizit unterschieden werden, ob alle Einträge auf dem Server gelöscht wurden oder ob lediglich der Server nicht erreichbar ist.

## 4.3 Bereitstellung zusätzlicher Einstellungsmöglichkeiten

Die zwei GroupDAV-Server Adressen sowie die Optionen Server- und Phone Wins, werden in Gevil-Sync über einer Activity vom Benutzer festgelegt. Aus der selben Activity kann auch der Offene Konflikte Dialog gestartet werden.

Um Gevil-Sync das Android typische Look-and-Feel zu geben, werden die Einstellungen in einer PreferenceActivity [Kal] vorgenommen. Die Elemente einer solchen werden in einer XML Datei spezifiziert. Zur Eingabe der beiden GroupDAV Server URLs, für Kalender- und Kontakte, wurden zwei `EditTextPreferences` genutzt. Um ein Scheitern der Synchronisation auf Grund eines Tippfehlers in den URLs zu vermeiden, wird bei Veränderung des Textes einer der beiden `EditTextPreferences` versucht, den Server unter der eingegebenen Adresse zu erreichen. Das Versuchsergebnis schreibt Gevil-Sync dann in die Description der `EditTextPreference`. Zur Auswahl zwischen Server Wins und Phone Wins fand jeweils eine `CheckBox Preference` Einsatz und über eine `Custompreference`, welche einem Button gleicht, gelangt der Benutzer zu der Activity, die alle offenen Konflikte anzeigt.

Um die von einem Benutzer getätigten Einstellungen auch über die Lebenszeit der App hinaus zu speichern, legt Gevil-Sync alle getätigte Einstellungen in SharedPreferences ab. Die Speicherung erfolgt für jeden Benutzer getrennt und wird beim Öffnen des Einstellungen Dialoges für den aktiven Benutzer ausgelesen. Auch bei Ausführung einer Synchronisation werden die Einstellungen aus den SharedPreferences des Sync Adapter Benutzernamens gelesen.

Daten, die in einer solchen Preferences Activity eingegeben werden, speichert Android auch ganz automatisch im Anwendungsspeicher. Jedoch wird dabei nicht zwischen verschiedenen Benutzernamen unterschieden. Diese von Android automatisch gespeicherten Daten finden in Gevil-Sync als Standardwert Einsatz. Hat der Benutzer beispielsweise bei einem vorherig konfigurierten Account bereits die Adresse seines GroupDAV Servers eingegeben, wird diese somit als Standardwert genutzt [deva].

Per xml erstellte Preference- Steuerelemente können auch direkt im Sync Adapter dargestellt werden. Durch den Eintrag `android:accountPreferences="@xml/preferences"` in der xml Datei, die den Authenticator des Sync Adapters beschreibt, in Gevil-Sync heißt diese authenticator.xml, muss dazu auf die preferences.xml verwiesen werden. Dies kann in der Praxis nur für `CheckBoxPreferences` verwendet werden. Beim Versuch, ein Fenster, für beispielsweise das Texteingabefeld, anzuzeigen, stürzt Android ab [sio]. Bei Klick auf die Checkbox zur Auswahl zwischen Server- und Phone Wins, wird durch die im Hintergrund ablaufenden Activity Klasse das aktuell ausgewählte Vorgehen als Beschreibung der Checkbox gesetzt. Wenn der Nutzer eine der beiden Server Adressen verändert, wird deren Erreichbarkeit geprüft. Diese Klasse wird bei der Einbindung im Sync Adapter ersetzt, wodurch die beiden beschriebenen Vorgänge nicht mehr ausgeführt werden. Aus diesen Gründen wurde auf die Einbindung der Optionen direkt im Android Accounts Menü verzichtet und der Einstellungen Dialog kann nur durch Starten der Gevil-Sync App erreicht werden.

## 4.4 Known Issues

Teilweise wurden bei der Implementierung von Gevil-Sync Fehler in eingebundenen Bibliotheken aufgedeckt. Um Gevil-Sync trotzdem lauffähig zu machen, wurden Workarrounds eingebaut, die das Problem jedoch nicht an der Wurzel beheben. Um diese Konstrukte, zum Beispiel bei Veröffentlichung eines Patches für eines der Probleme, schnell wieder zu finden, wurden diese in der Klasse `WorkarroundExternalBugs` zusammengefasst.

### 4.4.1 Ical4j-Vcard

Ical4j-Vcard wirft eine Exception, falls der letzte der durch Kommas getrennten Parameter leer ist. Nun hat jedoch kein aus Android gelesener Kontakt einen Namenssuffix, weshalb das betreffende Feld so gut wie immer leer ist. In den Workarrounds wird an dieser Stelle ein String eingefügt, welcher beim späteren Einlesen einfach ignoriert wird. VCards, die vom GroupDAV Server Citadel kommen, enthalten Einträge oft doppelt oder mehrfach. Solche doppelten Zeilen werden in den Workarrounds entfernt.

### 4.4.2 Jdom Android Fork

Jdom 1.1.1 hat in Zusammenarbeit mit Android Version 2.1 und früher einen Bug, der zum Absturz führt, wenn in einer xml Datei ein leerer Namespace vorkommt. [Jdo01] bietet eine gepatchte Version von Jdom 1.1.1 an, welche auch in Gevil-Sync Einsatz findet.

### 4.4.3 Löschen von Wiederholungsregeln unter Android

Sobald man im Android Kalender die Recurrence Rule eines sich wiederholenden Events entfernt, löscht Android das Event komplett aus dem Adressbuch und fügt es neu ein. Das heißt, dass die `Android _id` und die UID verloren gehen. In den meisten Fällen erkennt Gevil-Sync dies dann auch als Delete und Insert, sodass die Synchronisation fehlerfrei abläuft. Falls das betreffende Event jedoch auch auf dem Server verändert wurde, kommt es zu einem Delete-Update Konflikt, woraufhin das Delete verworfen wird. Das Resultat ist, dass das von Android neu erstellte Event auf dem Server und das vermeintlich gelöschte Event unter Android wieder eingefügt wird. Faktisch ist das Event dann doppelt vorhanden und die vorgenommenen Änderungen jeder Seite sind in je einem der beiden Duplikate vorhanden.

## 4.5 Future Work

In diesem Abschnitt werden Vorschläge beschrieben, durch die Gevil-Sync weiter entwickelt werden könnte. Da ihre Umsetzung den Rahmen dieser Arbeit überschritten hat, wurden sie lediglich hier zusammengefasst.

### 4.5.1 Wiederherstellungsfunktion

Mit einer Wiederherstellungsfunktion wäre es möglich, eine Synchronisation rückgängig zu machen. Somit könnten falsche Eingaben, auch nachdem sie synchronisiert wurden, wieder rückgängig gemacht werden. Dazu müsste, vor Ausführung jeder Operation, der unveränderte Kontakt in einem Backup gespeichert werden. In diesem Backup könnte dann mittels einer Suchfunktion, Informationen wie Name oder auch Metadaten wie Emailadresse gesucht werden und für einen dort gefundenen Kontakt, von Gevil-Sync gespiegelte Operationen auf beiden Seiten der Synchronisation wieder rückgängig gemacht werden. Der Speicherbedarf des Backups könnte beschränkt werden, indem Wiederherstellungsinformationen nur eine vom Nutzer definierbare Zeit lang aufbewahrt werden. Eine weitere Möglichkeit, den Speicherbedarf zu beschränken, wäre die Festlegung einer Speicher Obergrenze. Wenn Beispielsweise 3 Megabyte an Backup Informationen vorliegen, dürfte der Speicherbedarf nicht mehr erhöht werden. Dies könnte realisiert werden, indem für jede neue Sicherung die älteste aus dem Backup entfernt wird.

### 4.5.2 Performanceoptimierung mittels Android Dirty Flag

Im Android Adressbuch gibt es einen Integer Flag namens Dirty. Laut der Beschreibung von [deva], ändert sich dieses immer, wenn der Kontakt von jemandem geändert wird, der `CALLER_IS_SYNCADAPTER` Query Parameter nicht gesetzt hat. Dadurch könnte man für einen betreffenden Kontakt, ähnlich wie beim HTTP Etag, lediglich die UID und die `Android_id` auslesen und würde sich die Zeit für den Vergleich des Kontaktes mit dem im Sync-Zustand sparen.

### 4.5.3 Hintergrundtask zur Anzeige von Notifications

Treten bei der Synchronisation nicht eindeutig auflösbare Konflikte auf, speichert Gevil-Sync eine Nachricht persistent im Handyspeicher. Durch Starten der Gevil-Sync App kann der User sich diese Notifications anzeigen lassen. Eine intuitivere Lösung wäre es, bei jedem Auftreten eines derartigen Konfliktes eine Notification anzuzeigen. Jedoch ist der Sync-Adapter kein GUI Prozess und kann daher keine Acitivities oder Notifications anzeigen. Eine Möglichkeit, dies trotzdem zu implementieren, wäre ein separat laufender Prozess, der zum Beispiel durch einen Broadcast Intent über das Auftreten einer Notification informiert wird und dies als Icon im Android Menu anzeigt.

# 5 Evaluation

Um das Ergebnis der Implementierung von Gevil-Sync auszuwerten, wird in diesem Kapitel seine Arbeitsgeschwindigkeit, sowie bei Synchronisation verursachtes Datenvolumen gemessen. Die Ergebnisse sind dabei grafisch dargestellt und werden mit anderen Programmen verglichen.

## 5.1 Methologie

Zur Ausführung der Benchmarks wurde wieder das Desire HD mit Android 2.3.3 genutzt. Als GroupDAV Server wurde Citadel [Cit] eingesetzt. Dies lief in einer VMware Virtual Machine, welche unter Windows 7 ausgeführt wurde. Im eingesetzten Rechner arbeitete ein Core 2 Quad q9550, der mit 3,44Ghz arbeitete und 4GB Ram zur Verfügung hatte. Das Desire HD war über Wlan mit einem Router verbunden, an welchem der Server-Rechner per Kabel angeschlossen war.

Der für die Evaluation entwickelte Sync-Generator erstellt zufällige Einträge, die er ins Adressbuch beziehungsweise den Kalender einfügt. Die Methode `generateLuckyContact()` in der Klasse `GenerateContacts` ist für deren Generierung zuständig. Dazu erstellt die Methode zuerst eine `GevilVCard`, der sie als Vor- sowie als Nachname jeweils eine Zufallszahl zwischen 0 und 100.000 zuweist. Als nächstes werden Emailadressen eingefügt. Im ersten Schritt ist die Wahrscheinlichkeit, dass eine zufallsgenerierte Adresse eingetragen wird 80%. In 20% der Fälle wird gar keine Emailadresse eingetragen. Nun wird in jedem Schleifendurchlauf mit sinkender Wahrscheinlichkeit eine Emailadresse hinzugefügt. Als Adressen werden auch hier Zufallswerte genutzt. Im Ergebnis folgt die Vorkommenshäufigkeit von Emailadressen einer Zipf-Verteilung [Wzi]. Beim Einfügen von Telefonnummern ist das Vorgehen dasselbe, sodass zufällig zwischen keiner und mehreren Nummern je Kontakt eingefügt werden. Durch die Zipf-Verteilung entstehen Kontakte, die zwischen keiner und drei Telefonnummern und Emailadressen haben. Ein Kontakt mit mehr als drei Emailadressen konnte niemals beobachtet werden.

Die zufällige Generierung von Kalenderevents ist Aufgabe der Methode `generateLukcyEvent()` in der Klasse `GenerateEvents`. Wie beim Namen in den Kontakten werden Felder, die genau einmal vorkommen, mit einem Zufallswert ausgefüllt. Bei 50% aller Events wird eine Wiederholunsregel hinzugefügt. Um ein zufälliges Start und Enddatum zu generieren, werden zwei Zufallszahlen zwischen -30 und 30 herangezogen. Im nächsten Schritt wird zweimal das aktuelle Datum ausgelesen und jeweils einer der beiden Zufallswerte in Tagen addiert. Der kleinere der beiden so generierten Datum und

Uhrzeit Objekte ist das Anfangsdatum des Zufallsevents, der größere das Enddatum. Durch dieses Vorgehen sind die zufällig generierten Termine nicht über Jahrhunderte im Kalender verstreut, sondern alle innerhalb von drei Monaten aufzufinden. Dies hat den Vorteil, dass der Testausführende einen schnellen Überblick über die generierten Termine im Kalender gewinnt. Andernfalls könnten durch übersehene Termineinträge, die jedesmal mitsynchronisiert werden, die Testergebnisse verfälscht werden.

Um eine Synchronisation mit den gewünschten Operationen zu verursachen, wurde der Sync-Generator entwickelt. Der Verständlichkeit zuliebe beginnen wir an dieser Stelle mit dem einfachsten, nicht mit dem ersten der durchgeführten Tests. Dieser ist in Abschnitt 5.6 zu finden. Hier wurde die Dauer von Insert Operationen getestet. Dazu wurden n zufällig generierte Einträge auf dem Handy eingefügt, die Synchronisation gestartet und gemessen wie lange es dauert, bis die n Server Insert Operationen erkannt und ausgeführt waren. Server Inserts bezeichnen hier Operationen, die von der Synchronisationslogik auf dem Server durchgeführt werden, weil der User sie auf dem Handy erstellt hat. Im Gegensatz dazu entstehen Handy Inserts, wenn der User Einträge auf dem Server erstellt.

Bei Test 5.4 wurde der durch den Synchronisationsvorgang entstandene Traffic gemessen. Dabei wurde mit dem Programm Netcounter [url] das verbrauchte Datenvolumen vor und nach Synchronisationsausführung gemessen. Die Differenz wurde durch die Anzahl der getätigten Insert Operationen geteilt, um den mittleren Traffic zu erhalten.

Um die von Gevil-Sync erreichte Performance besser einordnen zu können, wurden in Abschnitt 5.5 seine Geschwindigkeit mit der von Google's Kalender Sync Adapter Implementierung verglichen. Getestet wurde jeweils ein Insert, ein Update und ein Delete, welche in der Testzeit von beiden Implementierungen gefunden und ausgeführt werden mussten. Wie vorgegangen wird, dass Google's Sync Adapter und Gevil-Sync diese Operationen erkennen und ausführen, ist in Abbildung 5.1 dargestellt.

Fangen wir der Verständlichkeit zuliebe mit dem Fall n=1 in Abbildung 5.1 an. Zu Beginn fügt der Sync-Generator zwei Einträge im Handy ein, woraufhin eine Synchronisation gestartet wird. Die Synchronisation erkennt die beiden neuen Einträge als Insert und führt die zwei Operationen auf dem Server aus. Diese sind in Abbildung 5.1 durch die beiden Wölkchen bei Schritt 1 dargestellt. Erst jetzt ist es uns möglich, durch Veränderung der Einträge auf dem Smartphone ein Update und ein Delete zu simulieren. Zusätzlich fügen wir noch einen weiteren Eintrag ein, sodass auch noch ein Insert im Performancetest vorkommt. Die drei Wölkchen bei Schritt 2 zeigen die Ausführung des erkannten Inserts, Updates und Deletes. Im 3. Schritt löscht der Sync-Generator alle Einträge auf dem Handy und durch Ausführung einer weiteren Synchronisation werden diese Deletes auch auf den Server weiter propagiert. Das so geleerte Adressbuch beziehungsweise der Kalender können nun für den nächsten Testdurchlauf genutzt werden. Bei der Synchronisationsausführung in Schritt 2 werden also 3n Operationen vom jeweiligen Sync Adapter erkannt und ausgeführt. Bei Schritt 1 und Schritt 3 sind es nur 2n Operationen. Bei Testausführung wird der Wert von n für jeden Benchmark Schritt inkrementiert.

Durch den Sync-Generator Einträge auf den Google Servern verändern zu lassen wäre, falls überhaupt möglich, mit viel Aufwand bei der Implementierung verbunden. Da bei

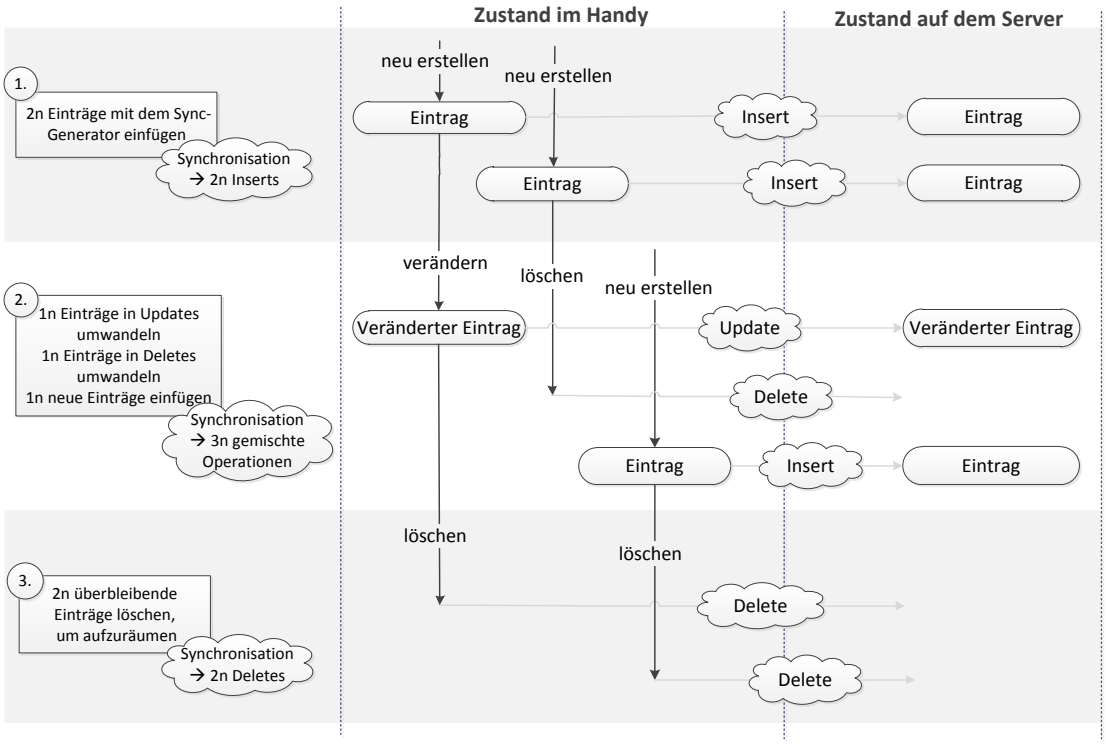


**Abbildung 5.1:** Vorgehensweise des Benchmarks beim Vergleich mit Google

Test 5.2 ausschließlich der GroupDAV Server Citadel Verwendung fand, war es möglich, Einträge auch serverseitig vom Sync-Generator verändern zu lassen. Somit konnte das im Vergleich mit Google eingesetzte Testverfahren erweitert werden und neben den Server-Inserts Updates und Deletes auch Handy- Inserts, Updates und Deletes erzeugt werden. Obendrein war die Bildung einer Konfliktsituation möglich. Eine solche wurde als zwei Operationen gezählt, weil der von der Konfliktbereinigung erstellte Eintrag auf Server und Client als Update Operation durchgeführt wird, wonach der in Konflikt stehende Eintrag auf beiden Seiten wieder identisch ist.

Um die Messungenauigkeiten bei allen Geschwindigkeitstests zu würdigen, wurde jeder Test drei mal durchgeführt und der Mittelwert der Ergebnisse herangezogen. Zusätzlich wurde noch die Standardabweichung der drei Ergebnisse berechnet.

## 5.2 Performance bei der Ausführung von Operationen

Im Performancetest wurden Server- sowie Handy- Insert-Updates und Deletes getestet. Die Anzahl der simulierten Operationen ergibt sich hier zu:

Operationen = Server Insert + Server Updates + Server Deletes + Handy Inserts + Handy Updates + Handy Deletes + 2*Konflikte

mit n = Server Insert = Server Updates = Server Deletes = Handy Inserts = Handy Updates = Handy Deletes = Konflikte

woraus folgt: Operationen = 8*n

Normalerweise läuft die Synchronisation als Hintergrundtask in Android ab. Für diesen Benchmark ist dies jedoch ungeeignet, da die Durchführungsdauer dann durch die Aktivitäten anderer Dienste stark beeinflusst wird. Deshalb wurden alle Geschwindigkeitsmessungen in einer Actitivity ausgeführt, welche vom Android Betriebssystem die höchste Priorität zugewiesen bekommt. Das Testen der Synchronisation von Kalenderevents funktionierte prinzipiell gleich. Auch hier kamen die Operationen Insert, Update, Delete und ein Konflikt zum Einsatz und ihre Anzahl wurde Schritt für Schritt erhöht. Die Dauer der Ausführung wurde festgehalten, indem zu Beginn die aktuelle Systemzeit in Millisekunden ausgelesen wurde und nach Abschluss der Synchronisation nochmals. Die Laufzeit der Synchronisation wurde als Differenz der beiden Werte ermittelt.

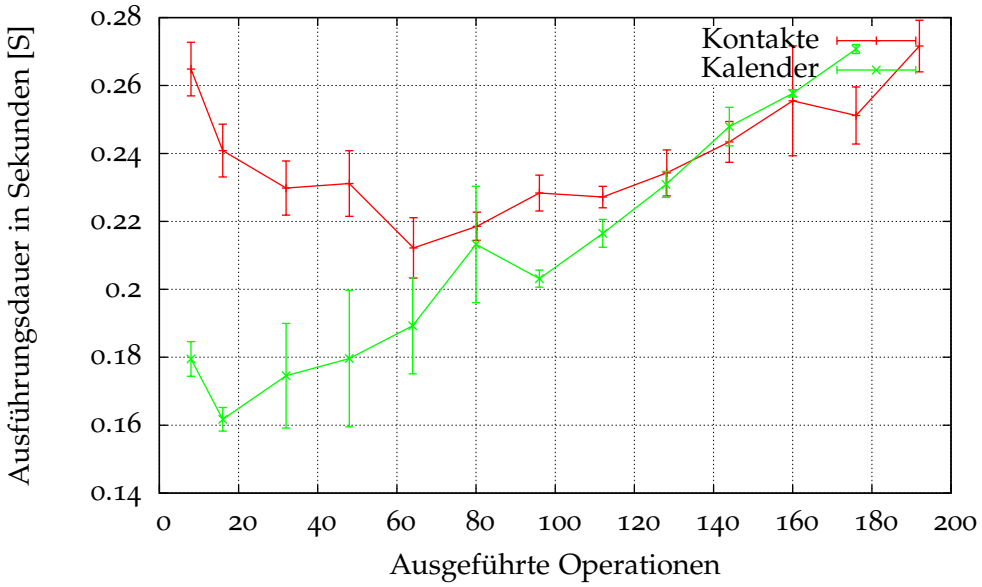


**Abbildung 5.2:** mittlere Ausführungsdauer pro Erkennung und Ausführung einer Operation in den Kontakten und dem Kalender

Abbildung 5.2 zeigt das Ergebnis der Testreihe. Getestet wurden 8n Operationen mit n=2,4,8,10,... Die Messpunkte liegen daher bei 16, 32, 48, 64,... Einträgen (im Adressbuch oder Kalender). Die einzelnen Messpunkte wurden mit Linien verbunden. Da die Messergebnisse Schwankungen unterlagen, wurde zusätzlich die Standardabweichung als horizontaler Balken an jedem Messpunkt eingetragen. Die Länge jedes Balkens entspricht dabei der

Standardabweichung. Im Test benötigten acht Operationen im Durchschnitt 2,118 Sekunden, also $\frac{2{,}118\ Sekunden}{8\ Operationen} = 0,265 \frac{Sekunden}{Operation}$ mittlere Ausführungsdauer.

Das Ergebnis der Messung zeigt, dass die Ausführungszeiten pro Operation zwischen 0,16 Sekunden und 0,27 Sekunden schwankten. Insgesamt ist das Wachstum etwas schlechter als linear, dieser Effekt ist jedoch gering und wird durch die Stauchung des Graphen übermäßig hervorgehoben. Bei der Kontaktsynchronisation sinkt die durchschnittliche Ausführungszeit je Kontakt anfänglich sogar mit der Anzahl der Operationen, was man darauf zurückführen kann, dass alle Operationen, die im Android Adressbuch ausgeführt werden als Batch angesammelt werden. Da die Gesamtlaufzeit der Synchronisation auch bei Tests mit sehr vielen Kontakten bei circa einer Minute lag, haben wir gezeigt, dass die Synchronisation in der Praxis flott arbeitet. Selbst wenn jemand 150 oder mehr Kontakte beziehungsweise Kalendereinträge abgleichen will, sollte eine runde Minute für die initiale Synchronisation ausreichen.

## 5.3 Performance beim Abgleich ohne Änderungen

Im praktischen Einsatz kommt eine Synchronisation mit vielen Operationen eher selten vor. Im Normalfall müssen zwar alle Einträge auf Änderungen hin überprüft werden, verändert wurden aber die wenigsten. Deshalb wird in diesem Benchmark getestet, wie lange die Synchronisation dauert, wenn dabei keine Operationen erkannt und ausgeführt werden.

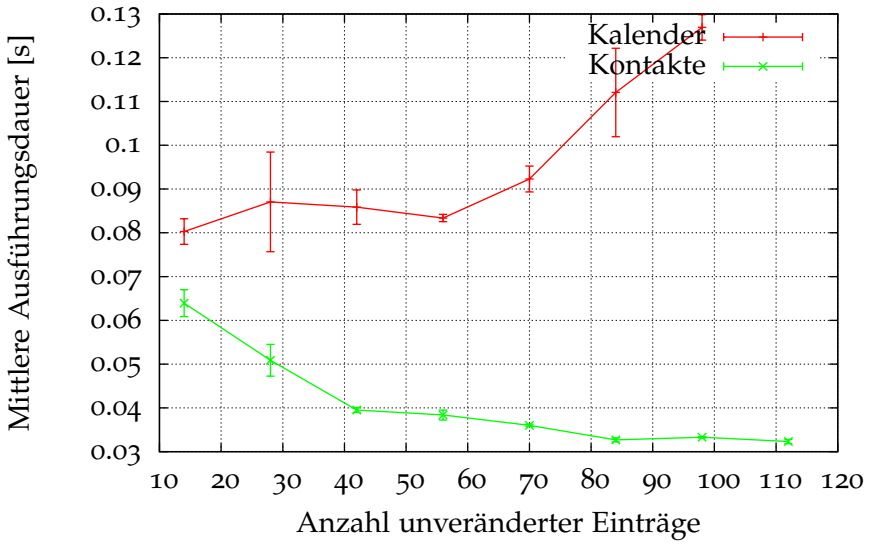


**Abbildung 5.3:** Synchronisationsdauer, wenn nur unveränderte Einträge vorhanden sind und deshalb keine Operationen erkannt und ausgeführt werden

Auch bei dieser Messung wurde in jedem Schritt die Anzahl der Einträge inkrementiert. Wenn man bedenkt, dass unveränderte Einträge auf Grund des Etags nicht vom Server geladen werden, jedoch jeder Eintrag aus Android ausgelesen und mit dem Sync-Zustand verglichen wird, kommt man zu dem Schluss, dass der hier notwendige Zeitaufwand primär dem Auslesen aus dem jeweiligen ContentProvider plus dem Zeitverbrauch der Vergleichsoperationen zuzurechnen ist. Es lässt sich also schließen, dass das Lesen von Kontakten durchweg etwas schneller als das Lesen von Kalenderevents ist. Da gelesene Kontakte in eine VCard und gelesene Kalenderevents in einen VCalendar dargestellt werden, muss auch dieser Vorgang in die Zeit des Einlesens mit eingerechnet werden. Dass die gemessene Synchronisationsdauer von Kalenderevents jedoch stärker steigt als die von Kontakten, ist im Prinzip von den beiden Bibliotheken unabhängig.

Die gemessenen mittleren Ausführungszeiten sind niedriger als in Abbildung 5.2. Dies liegt auf der Hand, weil dort nicht nur Zustände verglichen werden, sondern Operationen erkannt, zwischengespeichert, auf Konflikte durchsucht und dann ausgeführt werden mussten. Unter Beachtung der eben aufgestellten These, dass die in Abbildung 5.3 gezeigte Synchronisationsdauer primär der Dauer des Auslesens zuzurechnen sind, kann man sich ein grobes Bild machen, wie lange die Synchronisationslogik unabhängig vom Lesen in Anspruch nimmt. Dadurch, dass es sich in Abbildung 5.3 vergleichsweise um geringere Zeitspannen als bei Abbildung 5.2 handelt, fällt die starke Steigung bei den Kalenderevents für das Ergebnis von Abbildung 5.2 nur wenig ins Gewicht.

## 5.4 Verursachter Traffic bei Synchronisation

In diesem Test wurde versucht, das von der Synchronisation verursachte Datenvolumen zu messen. Da Netcounter den gesamten Traffic, der über die Wlan Schnittstelle verursacht wird, mitzählt bestand die Gefahr, dass die Ergebnisse durch Internetkommunikation von Anwendungen im Hintergrund verfälscht werden. Um dies zu vermeiden, wurde die Internetverbindung während des Testes getrennt. Trotzdem wirken die ermittelten Standardabweichungszahlen zu hoch, um ein wirklich gutes Resultat ziehen zu können.

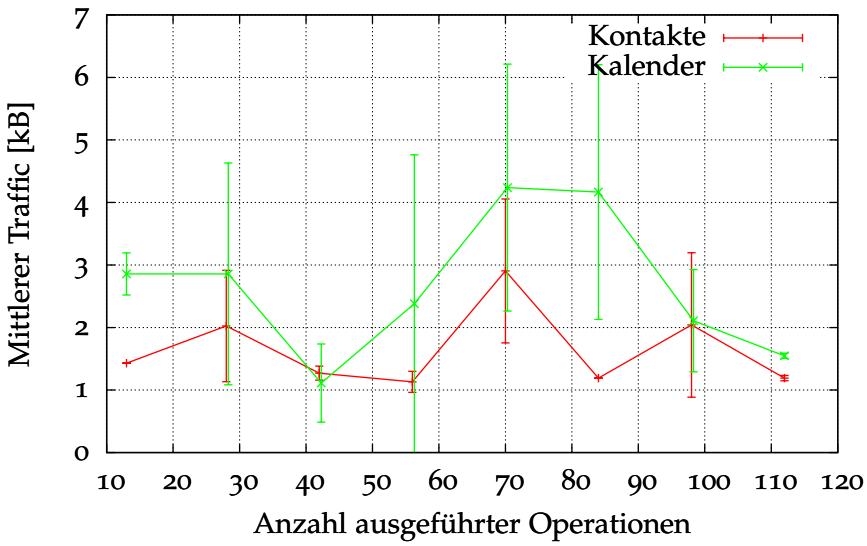


**Abbildung 5.4:** Von Gevil-Sync verursachte mittlere Netzwerkauslastung

Ausgeführt wurden bei diesem Benchmark Insert Operationen. Trotz der Hohen Standardabweichungen, kann man aus den gewonnenen Zahlen eine Vorstellung davon gewinnen, wie viel Datenvolumen ungefähr für die Synchronisation notwendig ist.

## 5.5 Performancevergleich mit dem Google Kalender

Die gemessene Synchronisationsdauer von Google's Kalender Sync Adapter ist aus zwei Gründen ungenauer als die Messergebnisse in denen Gevil-Sync alleine arbeitete. Da die Google-Synchronisation nicht aus dem Quellcode, sondern im Android Sync Adapter Menu gestartet wurde, musste die Dauer per Stoppuhr gemessen werden. Der zweite Faktor der einen negativen Einfluss auf die Genauigkeit der Google Ergebnisse hat, sind die unabänderlichen Schwankungen der Internet- und Server Auslastung. Neben diesen beiden Ungenauigkeitsfaktoren ist nicht mit letzter Sicherheit festzustellen, ob der Google Sync Adapter wirklich nur die erhofften Daten abgleicht, oder ob beispielsweise die Synchronisationsdauer anderer Daten ungewollt mitgezählt wurde. Ausschließliche Gevil-Sync Tests konnten in einer Activity anstatt eines Hintergrunddienstes ausgeführt werden, was derartige Effekte weitestgehend ausgeschlossen haben sollte.

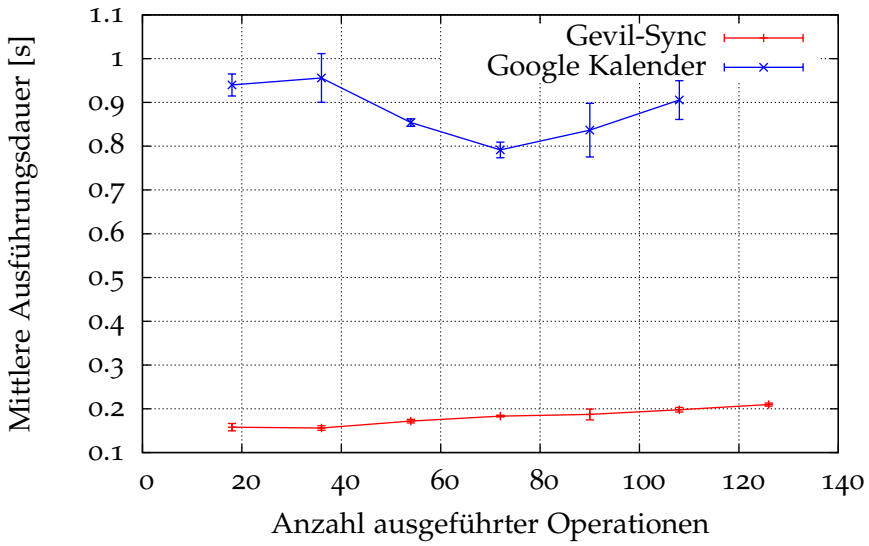


**Abbildung 5.5:** mittlere Ausführungsdauer je Operation von Gevil-Sync im Vergleich zur Google Kalender App

In diesem Vergleichstest macht Gevil-Sync einen durchweg positiven Eindruck, da die mittlere Ausführungsdauer in allen Messpunkten deutlich geringer als die des Google Sync Adapters war. Jedoch muss dabei beachtet werden, dass Gevil-Sync in einem lokalen Wlan arbeitete und einen unbeschäftigten Server als Kommunikationspartner hatte. Hingegen kommuniziert Google über das Internet mit seinen Servern, welche von einer möglicherweise stark schwankenden Anzahl von Benutzern mit genutzt wurden. Das Wachstum der beiden Funktionen unterscheidet sich nicht stark. Die mittlere Ausführungsdauer von Gevil-Sync steigt meist leicht an. Die Google Synchronisation unterliegt starken Schwankungen. Wenn man diese großzügig betrachtet, könnte man eine geringe Beschleunigung bei vielen Einträgen daraus ablesen. Dagegen spricht, dass durch die Messung per Stoppuhr zur Synchronisationsdauer von Google die Reaktionszeit vom Beenden der Synchronisation bis zum Anhalten der Stoppuhr hinzu kommt. Diese Reaktionszeit ist im Mittel unabhängig von der Anzahl der synchronisierten Events. Da der Ergebnisgraph die mittlere Ausführungsdauer zeigt, wird diese gleich bleibende Reaktionszeit jedoch durch eine stetig steigende Anzahl von Events geteilt. Durch diesen Effekt wird die mittlere Ausführungsdauer von Google's Sync Adapter in jedem Schritt verringert. Da die Reaktionsgeschwindigkeit der Person, welche die Stoppuhr bediente, nicht bekannt ist, kann auch nicht ermittelt werden, wie stark sich dieser Effekt auf das Ergebnis auswirkt.

Wie schon zuvor erläutert, musste zur Simulation der drei Operationen eine vorbereitende Operation, Schritt 1 in Abbildung 5.1, sowie eine abschließende Synchronisation, Schritt 3 in Abbildung 5.1, durchgeführt werden. Die Dauer dieser beiden Vorgänge wurde auf die gleiche Art ermittelt und wird in den zwei folgenden Graphen aufbereitet.

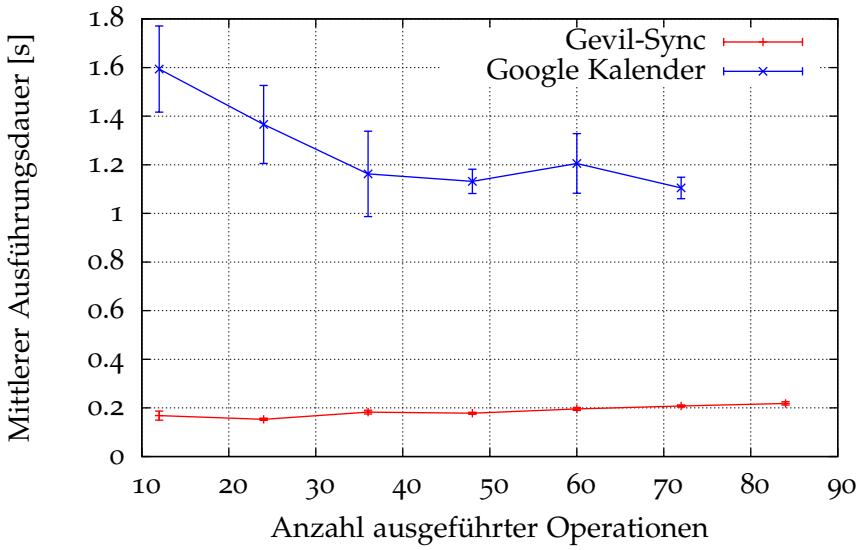


**Abbildung 5.6:** Vergleich der mittleren Ausführungsdauer bei 2n Insert Operationen

Auch beim Einfügen von Einträgen war die mittlere Ausführungsdauer von Gevil-Sync im lokalen Wlan durchweg schneller. Beim Wachstum ist Google hier wiederum im Vorteil. Aus Abbildung 5.6 lässt sich der Schluss ziehen, dass Google eine effizientere Übertragung der Einträge gewählt hat. Möglicherweise werden alle Einträge in einer einzelnen komprimierten Datei übertragen.

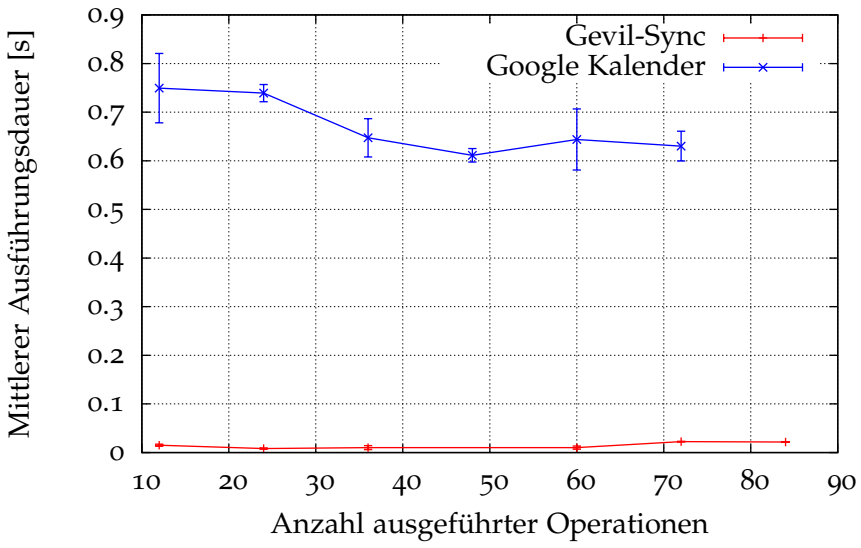


**Abbildung 5.7:** Vergleich der mittleren Ausführungsdauer bei 2n Delete Operationen

Abbildung 5.7 legt nahe, dass das Delete bei beiden Applikationen die schnellste Operation ist. Ansonsten können aus dem Graphen keine Erkenntnisse gezogen werden, die nicht schon aus den vorhergegangen gewonnen wurden.

# Literaturverzeichnis


[AnD] *AnDal - CalDAV Calendar for Android*. http://www.hypermatix.com/products/andal_caldav_calendar_for_android (Zitiert auf Seite 23)

[App] Apple: *iPhone in Unternehmen*. http://www.apple.com/de/iphone/business/integration/ (Zitiert auf Seite 23)

[Bla] Blackler, Jim: *Accessing the internal calendar database inside Google Android applications*. http://jimblackler.net/blog/?p=151 (Zitiert auf Seite 19)

[Cal] *Update Event on Phones Calendar. http://stackoverflow.com/questions/7656343/cant-update-event-on-phones-calendar-from-code* (Zitiert auf Seite 44)

[CD07] C. Daboo, L. D. B. Desruisseaux D. B. Desruisseaux: RFC: 4791, Calendaring Extensions to WebDAV (CalDAV). (2007), März. http://www.ietf.org/rfc/rfc4791.txt (Zitiert auf Seite 13)

[Cit] *Citadel Homepage*. http://www.citadel.org/ (Zitiert auf Seite 51)

[Dab11] Daboo, C.: RFC: 6352, CardDAV: vCard Extensions to Web Distributed Authoring and Versioning (WebDAV). In: *Internet Engineering Task Force (IETF)* (2011). https://tools.ietf.org/html/rfc6352 (Zitiert auf Seite 13)

[dat] *iCalendar spec: 4.3.5 Date- Time*. http://www.kanzaki.com/docs/ical/dateTime.html (Zitiert auf Seite 45)

[deva] *Developers.android Webseite*. http://developer.android.com (Zitiert auf den Seiten 44, 45, 47 und 49)

[devb] *developers.android.com Sample Sync Adapter*. http://developer.android.com/resources/samples/SampleSyncAdapter/index.html (Zitiert auf den Seiten 21, 22 und 39)

[DMF11] *DMFS-Homepage*. http://dmfs.org/. Version: November 2011 (Zitiert auf Seite 24)

[Dus07] Dusseault, L.M.: RFC: 4918, HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV). (2007), Juni. http://www.webdav.org/specs/rfc4918.html (Zitiert auf Seite 12)

[FD98a] F. Dawson, D. S.: RFC: 2445, Internet Calendaring and Scheduling Core Object Specification(iCalendar). In: *Network Working Group* (1998), November. http://www.rfc-editor.org/rfc/rfc2445.txt (Zitiert auf den Seiten 15 und 45)

[FD98b] F. Dawson, T. H.: RFC: 2426, vCard MIME Directory Profile. (1998), September. http://www.ietf.org/rfc/rfc2426.txt (Zitiert auf Seite 16)

[Fis07] Fischer, Reinhard: *Kalenderstandards im Internet*, Wirtschaftsuniversität Wien, Diplomarbeit, 2007 (Zitiert auf Seite 15)

[Goo] *Google Kalender*. http://de.wikipedia.org/wiki/Google_Kalender (Zitiert auf Seite 24)

[Hes04] Hess, H.: RFC: draft, Storage of Groupware Objects in WebDAV (GroupDAV). In: *OpenGroupware.org / KDE* (2004), November. http://svn.opengroupware.org/cgi-bin/viewvc.cgi/www/groupdav/trunk/draft-hess-groupdav-01.txt?diff_format=u&view=markup&pathrev=426 (Zitiert auf Seite 14)

[ica11] *sourceforge ical4j-vcard Download*. http://sourceforge.net/projects/ical4j/files/ical4j-vcard/. Version: 2011 (Zitiert auf Seite 18)

[Jdo01] *JDOM StringIndexOutOfBoundsException*. http://code.google.com/p/android-rome-feed-reader/. Version: 2001 (Zitiert auf Seite 48)

[jo08] jo: Funambol. In: *c't* (2008), Nr. 13 (Zitiert auf Seite 23)

[JWJ98] JAMES WHITEHEAD JR., MEREDITH W.: WEBDAV: IETF Standard for Collaborative Authoring on the Web. In: *IEEE INTERNET COMPUTING* (1998), September (Zitiert auf Seite 12)

[Kal] Kaløer, Mads: *Android Preferences*. http://www.kaloer.com/android-preferences (Zitiert auf Seite 47)

[LD] Lauren Darcey, Shane C.: *Working with the Android Calendar*. http://www.developer.com/ws/article.php/3850276/Working-with-the-Android-Calendar.htm (Zitiert auf Seite 19)

[LD05] Lisa Dusseault, Jim W.: Open Calendar Sharing and Scheduling with CalDAV. In: *IEEE INTERNET COMPUTING* (2005), März (Zitiert auf Seite 13)

[LN01] Lars Novak, Andreas J.: SyncML—Getting the mobile Internet in sync. In: *Ericsson Review* (2001), Nr. 3 (Zitiert auf Seite 14)

[Man03] Manhart, Klaus: Daten immer up to date. In: *www.tecchannel.de* (2003), Dezember (Zitiert auf Seite 14)

[Mod11] Modularity, QueBurt R. PashaZuck: *ical4j Wiki*. http://wiki.modularity.net.au/ical4j/index.php. Version: September 2011 (Zitiert auf Seite 17)

[NS] Nic Strong, Gareth H. Daniel McLaren M. Daniel McLaren: *Mobile device software from the creators of the Award Winning Symante Ghost*. http://www.spritesoftware.com/ (Zitiert auf Seite 23)

[RHA96] Roland H. Alden, Stephen J.: vCalendar The Electronic Calendaring and Scheduling Exchange Format Version 1.0. (1996), September. http://www.imc.org/pdi/vcal-10.txt (Zitiert auf Seite 15)

[sio] sionide21@gmail.com: *Cannot launch DialogPreference subclasses from accountPreferences window*. http://code.google.com/p/android/issues/detail?id=8350 (Zitiert auf Seite 48)

[Ste10] Steele, Sam: *Writing an Android Sync Provider: Part 1 and 2*. http://www.c99.org/2010/01/23/writing-an-android-sync-provider-part-1/. Version: 2010 (Zitiert auf Seite 21)

[url] https://market.android.com/details?id=net.jaqpot.netcounter (Zitiert auf Seite 52)

[Wec11] Wechselberger, Franz J.: *MyPhoneExplorer ist ein kostenloses Verwaltungs-Programm für Android-Smartphones und Sony-Ericsson-Handys*. http://www.chip.de/downloads/MyPhoneExplorer_16402327.html. Version: 2011 (Zitiert auf Seite 23)

[Wzi] *Zipf's law*. http://en.wikipedia.org/wiki/Zipf's_law (Zitiert auf Seite 51)

Alle URLs wurden zuletzt am 29.11.2011 geprüft.

**Erklärung**

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

---

(Martin Thielefeld)