

# Technische Hochschule Nürnberg

IT-Projekt  
Verteiltes Erdbebenwarnsystem  
Bearbeitungszeitraum: SS13 - WS13/14

## IT Projekt

vorgelegt von Christopher Althaus  
Baris Akdag  
Niklas Schäfer  
Benjamin Brandt  
Jürgen Hetzel

Betreuer Prof. Dr. Michael Zapf

Abgabe: 14. Februar 2014

## **Erklärung**

Hiermit versichern wir, dass wir die Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet zu haben.

---

Christopher Althaus

---

Baris Akdag

---

Niklas Schäfer

---

Benjamin Brandt

---

Jürgen Hetzel

Nürnberg, den 14. Februar 2014

## Abstract

Portable Geräte wie aktuelle Smartphones und Tablet-Computer besitzen in der Regel eine Vielzahl von Sensoren, darunter auch solche, die Beschleunigungen feststellen können. Diese können insbesondere genutzt werden, um Erschütterungen des Geräts festzustellen. Die unten gezeigte Grafik ist die Ausgabe einer Android-Anwendung (App), welche diese Sensoren ausliest.



Offensichtlich werden diese Sensoren ständig ausgelöst, wenn der Benutzer das Gerät mit sich führt, während er sich fortbewegt. Dabei sind die Werte der Sensoren unmittelbar von der individuellen Bewegung abhängig und daher stets zwischen zwei Geräten verschieden.

Interessant wäre es, wenn es möglich wäre, Korrelationen zwischen den Sensorwerten auf verschiedenen Geräten zu finden. Dies würde darauf hindeuten, dass beide Geräte, zumal wenn sie an verschiedenen Orten aufbewahrt werden, dasselbe Ereignis wahrgenommen haben, etwa eine Erschütterung im Boden.

Dies könnte dazu genutzt werden, um ein automatisches ErdbebenmeldeSystem zu realisieren. Wenn eine gewisse Menge von Geräten zur gleichen Zeit ein ähnliches Erschütterungsmuster detektieren, ist davon auszugehen, dass sich ein Erdbeben ereignet. Dies wird natürlich von den Anwendern selbst auch bemerkt werden, jedoch könnten die Geräte einerseits einen Alarm auslösen, der auch solche Menschen warnt, die aus diversen Gründen das Ereignis nicht wahrnehmen (schlafen oder im Auto sitzen), andererseits könnten Sicherheitsmaßnahmen in Gang gesetzt werden (automatisches Abstellen der Gasversorgung, Abstellen des Stroms an gefährlichen Orten usw.).

# Inhaltsverzeichnis

<b>1 Teamorganisation</b>	<b>5</b>
<b>2 Motivation (Christopher Althaus)</b>	<b>6</b>
<b>3 System Struktur (Christopher Althaus)</b>	<b>7</b>
<b>4 RESTful WebService (Baris Akdag)</b>	<b>9</b>
<b>5 User Interface (Niklas Schäfer)</b>	<b>10</b>
5.1 Info . . . . .	10
5.2 Device Map . . . . .	11
5.2.1 Google Map in ein Projekt integrieren . . . . .	11
5.2.2 User Interface . . . . .	13
5.2.3 Android Maps Extensions . . . . .	14
5.2.4 AsyncTask Marker hinzufügen . . . . .	15
5.2.5 Klassendiagramm DeviceMap . . . . .	17
5.2.6 Zukünftige Implementierung . . . . .	18
5.3 Settings . . . . .	18
<b>6 Erdbebenerkennung unter Android (Christopher Althaus)</b>	<b>19</b>
6.1 Abfragen der Sensordaten unter Android . . . . .	19
6.2 Erdbebenauswertung . . . . .	21
<b>7 Lokalisierung (Niklas Schäfer)</b>	<b>26</b>
7.1 Einleitung . . . . .	26
7.2 Location Provider des Android-Systems . . . . .	26
7.2.1 PASSIVE . . . . .	26
7.2.2 NETWORK . . . . .	27
7.2.3 Global Positioning System . . . . .	29
7.3 Positionsbestimmung in Android . . . . .	31
7.4 Umsetzung der Quakedetec App . . . . .	34
7.4.1 Erste Umsetzung und Probleme . . . . .	34
7.4.2 Finale Umsetzung . . . . .	35
7.4.3 Klassendiagramm Localizer . . . . .	38
<b>8 Kommunikation (Benjamin Brandt)</b>	<b>39</b>
<b>9 Testing (Benjamin Brandt)</b>	<b>43</b>
9.1 Lösungsansätze . . . . .	43

9.2 Durchführung . . . . .	43
<b>10 Ausblick</b>	<b>48</b>
<b>11 Fazit</b>	<b>48</b>

# 1 Teamorganisation

Die Projektgruppe besteht aus Niklas Schäfer, Baris Akdag, Christopher Althaus, Benjamin Brandt sowie Jürgen Hetzel. Innerhalb der Gruppe sind zu Beginn die verschiedenen Aufgabengebiete nach Interessen und Fähigkeiten des einzelnen verteilt worden.

Baris Akdag verfügte im Vorfeld über Fachkenntnisse in den Bereichen WebServices und Datenbanken. Er übernahm die Entwicklung des gesamten WebServices inklusive Datenbank und Bereitstellung.

Durch die Erfahrung von Christopher Althaus im Bereich Android Programmierung bot er sich neben Niklas Schäfer an, die Android Applikation zu entwickeln.

Dabei übernahm Niklas Schäfer als Hauptaufgaben die Lokalisierung der Geräte inklusive der Sicherstellung aktiver Standortbestimmung auf den Smartphones, sowie die Einbindung und Weiterentwicklung der Google Maps Karte. Weiterhin implementierte er die Einstellungen (Settings View und deren Funktion) und arbeitete am User Interface der App.

Christopher Althaus widmete sich neben der groben Strukturierung der App, hauptsächlich um die Aufgabengebiete rund um den Beschleunigungssensor und um die Benutzerbenachrichtigung im Falle eines Erdbebens. Diese Aufgaben umfassten zum einen die Erdbebenerkennung innerhalb der Applikation und zum anderen die Einbindung eines Diagramms zur Visualisierung der Beschleunigungsdaten.

Jürgen Hetzel und Benjamin Brand übernahmen während des Projektverlaufs einen Großteil der Literaturrecherche. Ebenso kümmerte sich Jürgen Hetzel zum Ende des Projekts um das Refactoring der Android Applikation. Da Benjamin Brand über eine große Auswahl von Geräten verfügte, übernahm er zusätzlich das Testen der Anwendung.

Über den gesamten Zeitraum der Bearbeitung ist eine enge Zusammenarbeit und gute Kommunikation Grundlage für ein erfolgreiches Umsetzen des Projekts gewesen.

## 2 Motivation (Christopher Althaus)

Sucht man im Internet Nachrichten über das Thema Erdbeben, wird schnell ersichtlich, dass bei nahe jeden Tag ein ernstzunehmendes Erdbeben auftritt. Sucht man weiterhin nach einer zuverlässigen Vorhersage für Erdbeben, stellt man ebenso schnell fest, dass dies zurzeit noch nicht möglich ist.

Mittels der Umsetzung eines verteilten Erdbebenwarnsystems ist die Warnung zwar auch erst möglich, wenn das Erdbeben bereits spürbar ist, da sich Erdbeben jedoch vom Epizentrum aus ausbreiten, können umliegende Bereiche noch von einer Warnung profitieren. Zudem könnten, wie bereits im Abstract beschrieben, Sicherheitsmaßnahmen in Gang gesetzt werden. Für das verteilte Erdbebensystem ist Android als Plattform ausgewählt worden. Dies begründet sich in der großen Verbreitung des Systems, welche momentan bei über 64% weltweit liegt<sup>1</sup>.

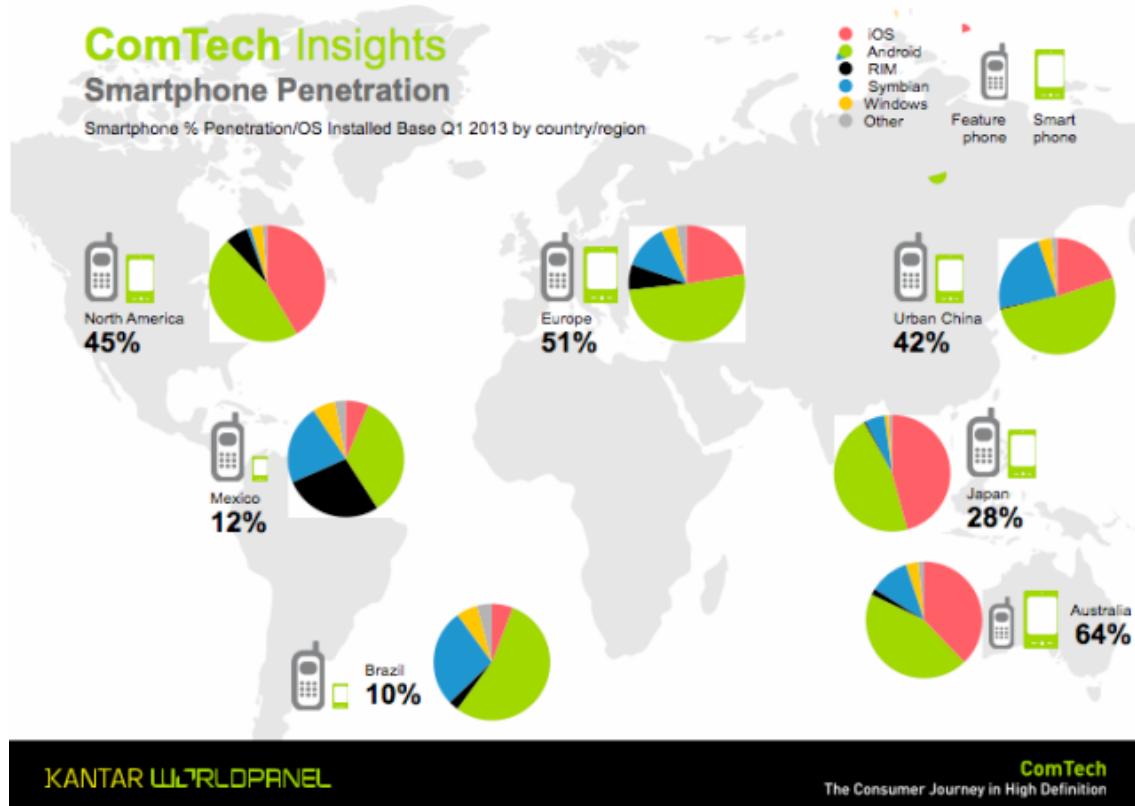


Abbildung 1: Smartphone Betriebssystem-Verbreitung

<sup>1</sup> <http://techcrunch.com/2013/04/28/android-picks-up-the-pace-in-smartphone-sales-over-ios-globally-while-windows-phone-continues-with-modest-gains-says-kantar/>

### 3 System Struktur (Christopher Althaus)

Die Erdbebenerkennung soll über ein verteiltes System erfolgen. Prinzipiell handelt es sich hierbei um ein Client-Server-System, wobei die Android Smartphones die Clients darstellen. Den Teil des Servers soll ein WebService übernehmen. Die die Strukturierung und Kommunikationsbeziehung dieser beiden Komponenten ist in Abbildung 2 dargestellt.

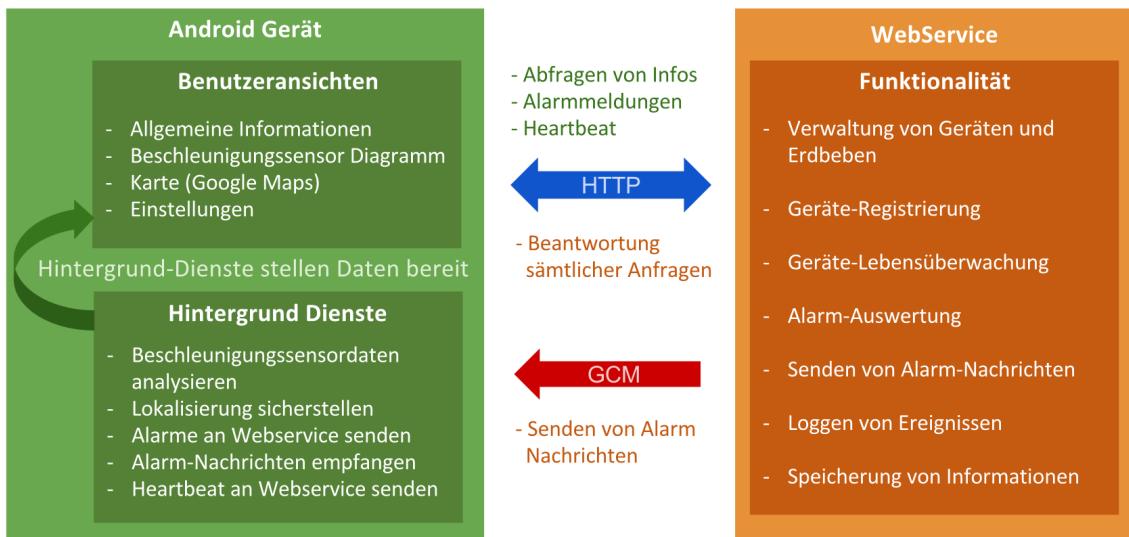


Abbildung 2: Struktur des Projektes

Das rechts dargestellte Android Gerät kann grundlegend in die Benutzeransichten und Hintergrunddienste unterteilt werden.

Die Benutzeransichten stellen dabei den für den Nutzer der App sichtbaren Teil dar. Hierzu gehören zum einen allgemeine Informationen, wie das letzte erkannte Erdbeben oder die Anzahl der verbundenen Geräte. Weiterhin soll dem Nutzer ein Diagramm angezeigt werden, welches die Daten des Beschleunigungssensors anzeigt, um dem Benutzer ersichtlich zu machen, dass die Sensorauswertung funktioniert. Ebenso soll innerhalb der Applikation eine Karte angezeigt werden, welche die verbundenen Geräte und die erkannten Erdbeben für den Nutzer visuell darstellt.

Um die Applikation für den Nutzer anpassbar zu machen, soll ein Einstellungsmenü existieren, in dem beispielsweise die Alarneinstellungen geändert werden können.

Neben diesen sichtbaren Anteilen bedarf es zahlreicher Hintergrundaktivitäten, welche die eigentlichen Aufgaben der Applikation bewerkstelligen. Zu den wohl Wichtigsten zählen hier die Lokalisierung des Gerätes und die Auswertung der Beschleunigungssensordaten. Diese Dienste sollen beim Systemstart des Smartphones automatisch im Hintergrund gestartet werden.

Weitere Aufgaben der im Hintergrund laufenden Dienste sind das Senden von Alarmen an den WebService, falls die Erdbebenerkennung meint, ein Erdbeben erkannt zu haben und das Empfangen und Verarbeiten von Alarm-Nachrichten des Webservices. Ebenso soll die Applikation in regelmäßigen Abständen eine Art Ping an den WebService schicken, damit dieser weiß, welche

Geräte noch aktiv sind.

Da die Hintergrunddienste Daten verwenden, welche auch in den Benutzeransichten benötigt werden, wie beispielsweise die Beschleunigungssensordaten für das Diagramm, sollen diese Daten für die Benutzeransichten bereitgestellt werden und somit nicht doppelt vom System abgefragt werden.

Der in der Abbildung 2 rechts dargestellte WebService dient somit der Verwaltung aller Geräte. Diese Verwaltung soll neben den Geräten auch die gemeldeten Erdbeben umfassen. Zur Verwaltung dieser Informationen soll eine Datenbank benutzt werden.

Die eigentliche Funktionalität aus Sicht des Android Gerätes, welche der WebService bereitstellt, ist die Geräte-Registrierung, die Geräte-Lebensüberwachung und die Alarmauswertung. Die Geräte-Registrierung soll dabei vom Nutzer unbemerkt beim ersten Aufrufen der App geschehen. Die Geräte-Lebensüberwachung dient, wie bereits beschrieben, dazu, dass der WebService Kenntnis darüber hat, welche Geräte noch aktiv sind. Wichtig wird dies bei der Alarmauswertung. Dabei soll nach einem eingehendem Alarm nach dem Mehrheitsprinzip ausgewertet werden, ob es sich um einen Fehlalarm handelt oder nicht. Da jedoch nur Geräte, welche meinen ein Erdbeben erkannt zu haben, eine Meldung an den WebService schicken, ist es wichtig zu wissen, welche Geräte in der Nähe noch aktiv sind, die keine Meldung geschickt haben.

Um in der Entwicklungsphase Fehler schneller erkennen zu können, soll innerhalb des WebService eingehende Ereignisse, wie beispielsweise eine Alarmnachricht eines Gerätes, geloggt werden. Wie in der Abbildung 2 ersichtlich, können das Android Gerät und der WebService mittels einer HTTP Verbindung bidirektional kommunizieren. Innerhalb der Android Anwendung sollen alle Anfragen, Alarmmeldungen und auch die Lebenszeichenüberwachung mittels HTTP an den Service geschickt werden und daraufhin auch mittels HTTP beantwortet werden.

Wie abgebildet, nutzt der WebService zur Alarmierung der Android-Geräte im Falle eines Erdbebens eine Verbindung namens GCM. GCM steht hierbei für Google Cloud Messaging und erlaubt es, Nachrichten an ein Android Gerät zu verschicken, ohne dabei eine extra Verbindung herstellen zu müssen.

## 4 RESTful WebService (Baris Akdag)

Todo.

## 5 User Interface (Niklas Schäfer)

Das User Interface der App besteht aus drei Bereichen:

- Info
- Device Map
- Settings

In der App wurde ein *ViewPager* implementiert, mit welchem es möglich ist, der App *Fragments* (Views) hinzuzufügen, die über Tabs im oberen Bereich der App oder per Wischgesten zu erreichen sind. In den ViewPager wurden ein *Info Screen* und eine *Device Map* integriert. Des Weiteren können über einen Button im oberen rechten Bereich die Einstellungen der App geöffnet werden.

### 5.1 Info

Die App bietet unter dem Tab Info eine Übersicht über die wichtigsten Daten. Im oberen Bereich befindet sich ein Graph, der die Daten des Beschleunigungssensors darstellt. Dabei zeigt er nicht alle Achsen an, die der Beschleunigungssensor ausliest, sondern nur einen Durchschnittswert aller Achsen. Für die Entscheidung nur einen Graph anzusehen, sprechen zwei Gründe. Zum Einen ist ein kontinuierliches Neuzeichnen sehr leistungshungrig, sodass die App auf einem performanceschwachen Gerät ruckeln und einfrieren würde, wenn mehrere Graphen ständig neu gezeichnet werden müssten. Zum Anderen kann es für einen Nutzer verwirrend sein, wenn mehrere stark schwankende Graphen dargestellt werden würden. Somit ist ein einzelner Graph eine ansprechendere, übersichtlichere und leistungsschonendere Lösung. Weiterhin kann der Nutzer die Anzahl der momentan verbundenen Geräte ablesen, was gleichzeitig auf die Zuverlässigkeit der Erdbebenerkennung schließen lässt (Stichwort: *Mehrheitsentscheid*). Darunter befinden sich Informationen zum letzten erfassten Erdbeben, wie Datum, Uhrzeit und Land/Stadt des letzten Bebens. Unter dem letzten Beben befindet sich die aktuelle Position in Form einer vollständigen Adresse. Außerdem wird im unteren Bereich des Screens der aktuell genutzte Ortungsdienst und die Genauigkeit des erfassten Standorts angezeigt (Weitere Informationen zum Ortungsdienst, siehe Kapitel 7.2 Location Provider des Android-Systems).



Abbildung 3: Info

## 5.2 Device Map

Da die Quakedetec App durch Mehrheitsentscheid darauf schließt, ob ein übersendeter Alarm wirklich aus einem Beben resultiert, ist die App umso verlässlicher, je mehr Geräte sich in der eigenen Umgebung befinden, die ebenfalls die App nutzen. Daher besitzt die App eine Google Map, in der der eigene Standort und der der anderern Nutzer angezeigt wird. Dadurch kann sich der Nutzer einen Überblick verschaffen, ob sich in der eigenen Umgebung weitere Nutzer befinden und kann daraus schließen, ob in seinem Gebiet die App zuverlässig arbeitet.

### 5.2.1 Google Map in ein Projekt integrieren

Um eine Google Map in einem Android Projekt verwenden zu können, muss dem Projekt die *Google Play Service SDK* hinzugefügt werden. Dazu wird das SDK installiert, als Android Application dem Workspace hinzugefügt und als Android Library im eigenen App Projekt referenziert. Außerdem muss noch folgender Eintrag im *AndroidManifest.xml* vorgenommen werden:

Listing 1: Google Map AndroidManifest.xml Eintrag

```
1 <meta-data  
2     android:name="com.google.android.gms.version"  
3     android:value="@integer/google_play_services_version" />
```

Weiterhin wird ein Android API Key benötigt. Um diesen zu erstellen, ist es notwendig ein API Projekt in der Google API Console anzulegen (Weitere Information dazu sind der Google Maps Dokumentation zu entnehmen). Mit Hilfe dieses API Projekts kann ein Android API Key generiert werden, indem der SHA-1 Fingerprint des eigenen Android Zertifikats und der Name des eigenen App Projekts durch Semikolon getrennt in der Google API Console eingetragen wird.

Listing 2: Android API Key generieren

```
1 BB:0D:AC:74:D3:21:E1:43:67:71:9B:62:91:AF:A1:66:6E:44:5D:75;com.th.nuernberg.itp.earthquakedetection
```

Der SHA-1 Fingerprint kann mit folgendem Befehl in der Shell unter Linux ausgegeben werden:

Listing 3: Fingerprint Ausgabe

```
1 keytool -list -v -keystore ~/.android/debug.keystore -alias androiddebugkey  
-storepass android -keypass android
```

```
Last login: Sat Dec 14 19:37:45 on ttys000  
macbook-nk:~ Niklas$ keytool -list -v -keystore ~/.android/debug.keystore -alias androiddebugkey -storepass android -keypass android  
Aliasname: androiddebugkey  
Erstellungsdatum: 17.04.2013  
Eintragstyp: PrivateKeyEntry  
Zertifikatskettenlänge: 1  
Zertifikat[1]:  
Eigner: CN=Android Debug, O=Android, C=US  
Ausssteller: CN=Android Debug, O=Android, C=US  
Seriennummer: 516dd88a  
Gültig von: Wed Apr 17 01:02:34 CEST 2013 bis: Fri Apr 10 01:02:34 CEST 2043  
Digitaler Fingerabdruck des Zertifikats:  
MD5: 99:D6:29:E4:81:7C:52:E0:3D:AE:A5:1D:AE:84:75:2A  
SHA1: B5:AB:D3:C4:A3:F5:0B:E0:5C:96:AF:C9:76:EC:BA:7F:2B:7F:BB:DB  
Unterschrift-Algorithmusname: SHA1withRSA  
Version: 3  
macbook-nk:~ Niklas$
```

Abbildung 4: SHA-1 Fingerprint

Nachdem die beschriebene Eintragung vorgenommen wurde, wird ein API Key generiert und angezeigt:

Key for Android apps (with certificates)	
API key:	AIzaSyAZ1RZUeVsn74xzulnJ3Xr-Y1suCcWomWE
Android apps:	6E:88:61:6D:34:9B:32:E3:AA:A9:22:6F:50:CE:D6:50:75:48:3B:1E;com.th.nuernberg.itp.earthquakedetection
Activated on:	May 27, 2013 9:27 AM
Activated by:	schaefe.rnk@googlemail.com

Abbildung 5: Google API Console nach erfolgreicher API Key Generierung

Daraufhin muss in der Google API Console der Service *Google Maps API v2* aktiviert werden.



Abbildung 6: Google Maps API Service aktivieren

Um die Konfiguration abzuschließen, muß der API Key noch der *AndroidManifest.xml* der App hinzugefügt werden.

Listing 4: Google Map API Key hinzufügen

```
1 <meta-data  
2     android:name="com.google.android.maps.v2.API_KEY"  
3     android:value="AIzaSyAZ1RZUeVsn74xzulnJ3Xr-Y1suCcWomWE"/>
```

Der letzte Schritt um die Google Map zu implementieren besteht darin, ein *SupportMapFragment* der entsprechenden *Android Activity (View)* hinzuzufügen, welche die Map darstellen soll.

Listing 5: SupportMapFragment hinzufügen

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <fragment xmlns:android="http://schemas.android.com/apk/res/android"
3         android:id="@+id/map"
4         android:layout_width="match_parent"
5         android:layout_height="match_parent"
6         android:name="com.google.android.gms.maps.SupportMapFragment"/>
```

### 5.2.2 User Interface

Die Google Map ist im Tab *Karte* zu finden. Sobald der Tab geöffnet wird, wird die Ansicht auf die eigene Position gesetzt, welche durch einen blauen Marker dargestellt wird. Tippt man diesen Marker an, öffnet sich ein Kontextmenü, welches die letzte Positionsaktualisierung und die Genauigkeit des Standortorts anzeigt. Auf der Map gibt es drei User Interface Elemente. Im rechten oberen Abschnitt befindet sich ein Button, der die Kamera zur eigenen Position führt. Wenn sich der Nutzer per Touchgesten durch die Map bewegt hat, kann er sich mit Hilfe dieses Buttons zurück zu seiner Position navigieren lassen. Am rechten unteren Rand hat der Benutzer die Möglichkeit in die Map hinein und heraus zu zoomen.

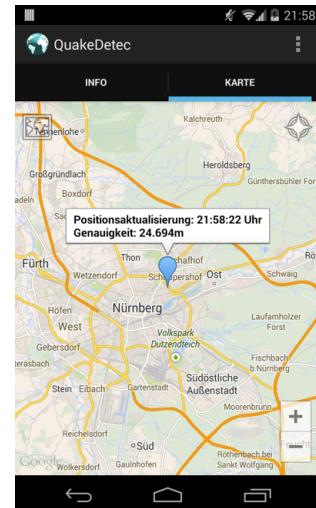


Abbildung 7: Map

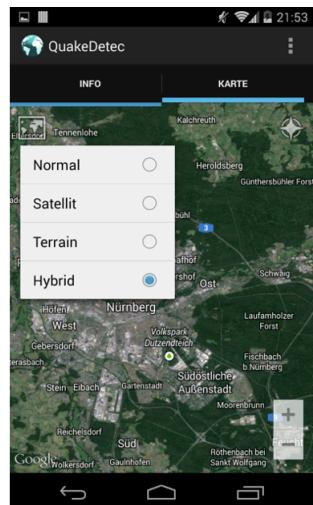


Abbildung 8: Map Types

Am linken oberen Rand ist ein Button zu finden, der die Möglichkeit bietet, die Ansicht der Karte zu ändern. Man kann zwischen *Normal*, *Satellit*, *Terrain* und *Hybrid* wählen. Der Button für den Zoom ist in der Google Map API enthalten. Die Buttons für die Ansicht und den eigenen Standort wurden eigenständig als Layout über die Google Map gelegt. Dies ist notwendig, da es in der API keine Funktion gibt, eine Möglichkeit für den Ansichtswechsel darzustellen. Die Funktion die Kamera auf den eigenen Standort zu bewegen, ist zwar enthalten, hat aber ungewünschte Nebeneffekte.

Um diese Funktion zu aktivieren, gibt es folgenden Methodenaufruf in der API:

```
1 googleMap.setMyLocationEnabled(true);
```

Dieser Aufruf bewirkt allerdings nicht nur, dass ein Button sichtbar wird, der die Kamera auf den eigenen Standort bewegt, sondern auch, dass die Google Map eigenständig Positionsdaten über die Smartphone Sensoren abruft. Dadurch steigt der Akkuverbrauch stark an, weil die Map auf alle verfügbaren Location Provider kontinuierlich zugreift (*GPS* und *NETWORK*). Des Weiteren hätte man dann zwei unterschiedliche Standorte, da die App einen Standort abruft und die Google Map ebenfalls einen Standort bestimmt. Somit würde gegebenenfalls in der Google Map ein anderer Standort angezeigt werden als der Standort, der in der Info Übersicht angezeigt wird. Um diese ungewünschten Nebeneffekte zu vermeiden, musste diese Funktion deaktiviert und neu nach den Ansprüchen der App implementiert werden.

Listing 6: "Kamera auf letzte bekannte Position setzen"

```

1  public void updateCameraToLastKnownLocation(int zoom)
2  {
3      if(this.googleMap != null && this.lastKnownLocation != null)
4      {
5          LatLng latLng = new LatLng(lastKnownLocation.getLatitude(),
6              lastKnownLocation.getLongitude());
7          CameraUpdate cameraUpdate = CameraUpdateFactory.newLatLngZoom(latLng,
8              zoom);
9          googleMap.animateCamera(cameraUpdate);
11     }
12 }
```

Die Methode aus Listing 6 wird ausgeführt, wenn der beschriebene Button gedrückt wird. Die Variable *lastKnownLocation* hält ein Objekt vom Typ *Location* und wird immer gesetzt und aktualisiert, wenn die App (*Localizer Klasse*) einen neuen Standort ermittelt hat.

### 5.2.3 Android Maps Extensions

Die Google Map API hat ein sehr starkes Performanceproblem, wenn auf der Map eine hohe Anzahl von Markern angezeigt wird. Ab einer Größenordnung von ca. 100000 Markern fängt sie sehr stark an zu ruckeln und friert teilweise ein. Daher wurde eine Library benutzt, die der Map eine Clustering Funktion hinzufügt. Diese Library ersetzt die Google Play Services Library und kann gleichwertig genutzt werden. Clustering bedeutet, dass Marker je nach Zoomlevel gebündelt oder wieder entbündelt werden. Je mehr aus der Map herauszoomt wird, desto mehr Punkte werden gebündelt. Wird wieder in die Map herein gezoomt, werden sie wieder entbündelt. Dadurch wird immer eine kleine Anzahl an Markern dargestellt und die Performance bleibt stabil auf einem guten Niveau. Gebündelte Punkte zeigen auf ihrem Label die Anzahl der zusammengefassten Punkte an.

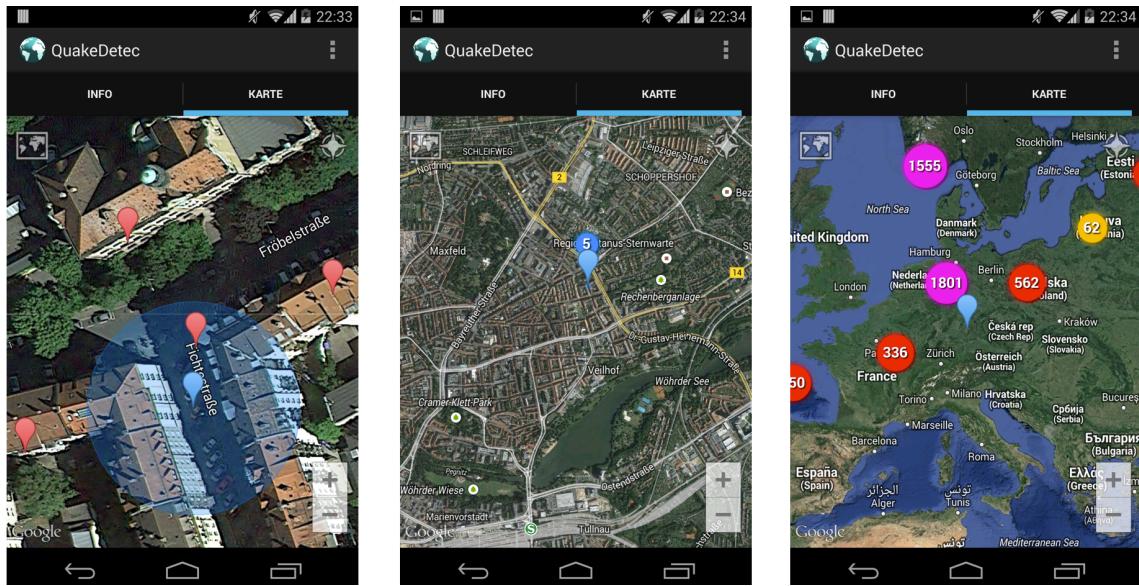


Abbildung 9: Android Maps Extensions

#### 5.2.4 AsyncTask Marker hinzufügen

Ein weiteres Problem der Google Maps API ist, dass ebenfalls das Hinzufügen von Markern auf der Map bei einer hohen Anzahl sehr lange dauern kann und für die Dauer dieses Vorgangs einfriert und unbenutzbar ist. Daher war es notwendig auch für dieses Problem eine alternative Implementierung zu entwickeln. Gelöst wurde das Problem durch einen AsyncTask, der die Marker der Map hinzufügt.

Listing 7: „Äsynchrone hinzufügen von Markern“

```

1  private class CreateAndAddMarkerTask extends AsyncTask<String, String, String> {
2
3      private ArrayList<MarkerOptions> markerOptionsList = new ArrayList<MarkerOptions>();
4
5      private ProgressBar progressBar;
6      private int progressStatus;
7      private ArrayList<LatLng> devicePosis;
8
9      public CreateAndAddMarkerTask(ArrayList<LatLng> devicePosis, ProgressBar progressBar, int
10         progressStatus)
11     {
12         this.devicePosis = devicePosis;
13         this.progressBar = progressBar;
14         this.progressStatus = progressStatus;
15         progressBar.setVisibility(ProgressBar.VISIBLE);
16     }
17
18 }
```

```

19     @Override
20     protected String doInBackground(String... params) {
21         int i = 0;
22         while(this.devicePosis.size() > 0 && i < 5000 && i < devicePosis.size())
23     {
24         MarkerOptions markerOptions = new MarkerOptions()
25             .icon(BitmapDescriptorFactory
26                 .defaultMarker(BitmapDescriptorFactory.HUE_RED))
27             .position(devicePosis.get(i));
28         markerOptions.title("Device");
29         markerOptionsList.add(markerOptions);
30         devicePosis.remove(i);
31
32         publishProgress("");
33         i++;
34     }
35     return null;
36 }
37
38     protected void onProgressUpdate(String... progress) {
39
40         progressStatus++;
41         progressBar.setProgress(progressStatus);
42     }
43
44     protected void onPostExecute(String str) {
45         for(MarkerOptions markerOptions : markerOptionsList)
46             googleMap.addMarker(markerOptions);
47         progressBar.setVisibility(ProgressBar.INVISIBLE);
48         this.cancel(true);
49         if(devicePosis.size() > 0)
50             new CreateAndAddMarkerTask(devicePosis, progressBar, progressStatus).execute();
51     }
52 }
```

Dieser *AsyncTask* bringt den Vorteil, dass die Map nicht mehr einfriert, wenn eine hohe Anzahl an Markern hinzugefügt werden muss. Zwar wird immernoch der gleiche Zeitraum beansprucht, die auch die bereitgestellte Funktion *addMarker(Marker marker)* aus der Google API benötigt, sodass auch mit dieser Lösung nicht alle Marker sofort dargestellt werden können, allerdings wird dem Nutzer eine *Progress Bar* angezeigt, welche im oberen Bereich unter der Tableiste sichtbar wird, solange der Vorgang aktiv ist. So kann der Nutzer nachvollziehen, ob und wann alle Standorte anderer Nutzer dargestellt werden. Außerdem ist zu beobachten, wie sich die Map während des Vorgangs mit Markern füllt, wodurch schon von Anfang an andere Nutzer auf der Map wahrgenommen werden können. Dies wäre mit der bereitgestellten Funktion aus der API auch nicht möglich, da die Map eingefroren wäre, bis der Vorgang abgeschlossen ist. Bei einer Anzahl von unter 10000

Markern liegt die Geschwindigkeit des Vorgangs noch unter einem für den Nutzer wahrnehmbaren Bereich. Erst ab ca. 50000 Markern, dauert der Vorgang etwa 5-10 Sekunden. In einer zukünftigen Implementierung wird diese Größenordnung nicht mehr überschritten (siehe Kapitel 5.2.6).

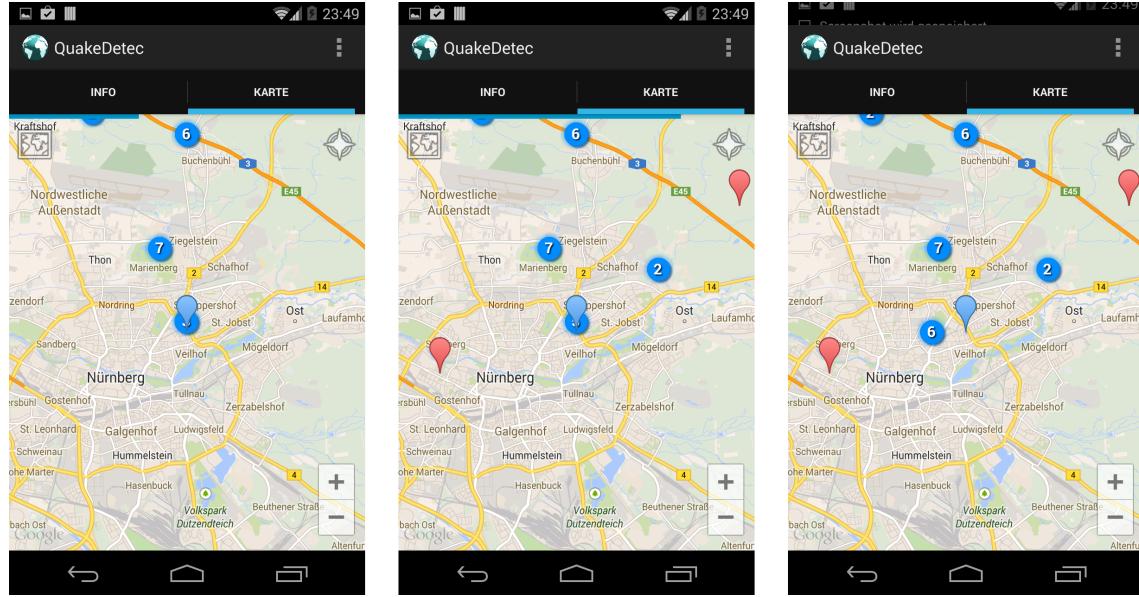


Abbildung 10: AsyncTask Marker hinzufügen

### 5.2.5 Klassendiagramm DeviceMap

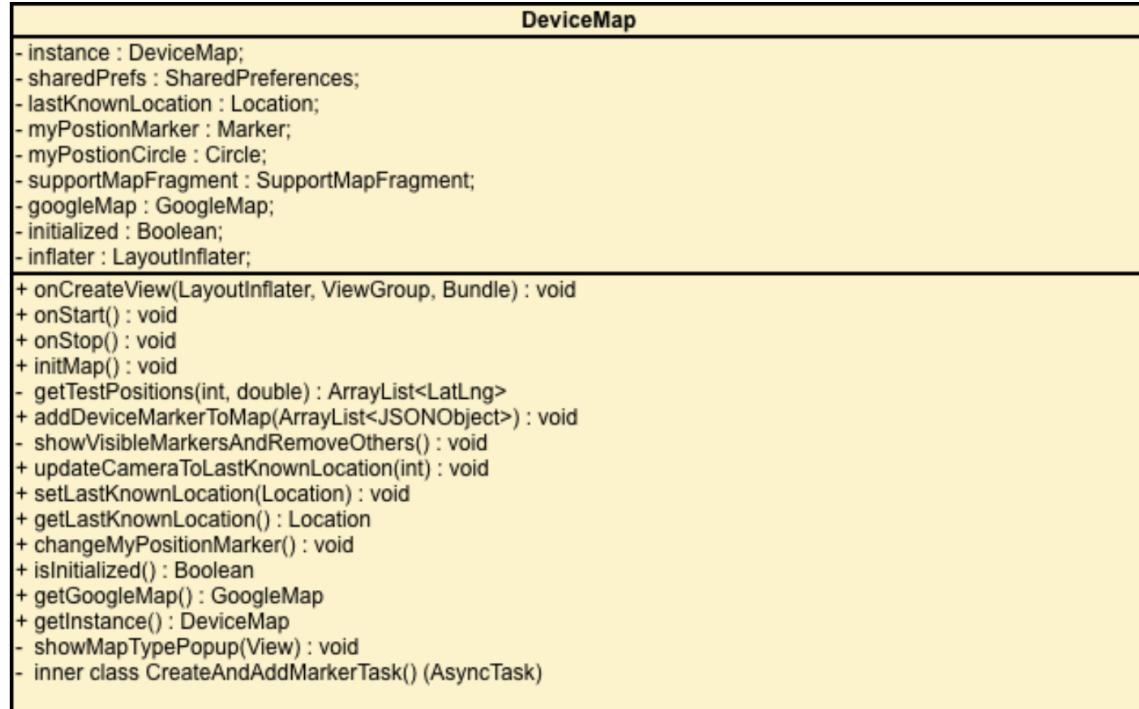


Abbildung 11: Klassendiagramm Device Map

### 5.2.6 Zukünftige Implementierung

Bei der aktuellen Implementierung kommen schnell Fragen zum Datenschutz auf. Zwar werden keine Informationen über andere angezeigte Geräte herausgegeben, aber wenn sich eine geringe Anzahl an anderen Geräten in der Umgebung befinden, kann ohne weiteres ein Bewegungsprofil erkannt werden. Wenn der Nutzer vielleicht sogar weiß, um wen es sich beim angezeigten Nutzer handelt (bspw. einen Freund, der ebenfalls die App nutzt), ist dies datenschutztechnisch sehr bedenklich. Daher sollen die Daten der anderen Geräte schon auf dem Server zusammengefasst werden und nur in zusammengefasster Form an die App ausgeliefert werden. Dazu soll ein gedachtes Raster über die Map gelegt werden. In jedem Rasterfeld werden alle Geräte zusammengefasst. Der zusammengefasste Punkt bekommt dann die Koordinaten, die sich im Mittelpunkt des Rasters befinden. An die App werden dann nur die gebündelten Punkte, Anzahl der Geräte jedes gebündelten Punktes und die Rastergröße weitergeleitet. Somit werden auf der Map keine genauen Punkte mehr angezeigt. Es wird weiterhin ein blauer durchsichtiger Kreis um die Punkte gelegt, der der Größe eines Rasterfeldes entspricht. Dadurch ist genau erkennbar, wie viele Geräte sich in einem bestimmten Gebiet/Raster befinden, ohne dass Bewegungsprofile eines Gerätes zu erkennen sind. Die Berechnung der gebündelten Standortdaten wird in bestimmten Zeitabständen durchgeführt und gecached, was dem Datenschutz durch die leicht zeitverzögerten Daten ebenfalls zuträglich ist. Da die Berechnung auf dem Server stattfindet, werden keine genauen Standortdaten mehr zwischen Server und App übertragen, wodurch ein Abgreifen der Standortdaten aus den http-Requests ebenfalls unterbunden wird. Bei dieser Implementierung wird die Rastergröße und das Berechnungsintervall frei konfigurierbar sein.

## 5.3 Settings

Im Hauptbereich der App befindet sich in der oberen rechten Ecke ein Button, über welchen die Einstellungen der App erreichbar sind. Dort kann der Nutzer Einstellungen der App vornehmen. Er hat beispielsweise die Möglichkeit die Standardeinstellung für den Map Type der Device Map zu ändern. Es kann zwischen den Typen *Normal*, *Terrain*, *Hybrid* und *Satellite* gewählt werden. Weiterhin gibt es die Möglichkeit die Notifications anzupassen. Es können jeweils Ton, Vibration und LED aktiviert/deaktiviert werden. Diese Einstellung wirkt sich auf sämtliche Notifications aus, die mit der App in Verbindung stehen, z.B. Erdbebenwarnung, deaktivierte Standortbestimmung, und so weiter. In der aktuellen Version befinden sich noch Einstellungen für den Server (IP und Port). Diese dienen dem Debugging und werden entfernt sobald die App veröffentlicht wird. Der letzte Punkt *Zurücksetzen* bietet dem Nutzer die Möglichkeit, die gesetzten Einstellungen wieder auf die Standardeinstellungen zurückzusetzen.

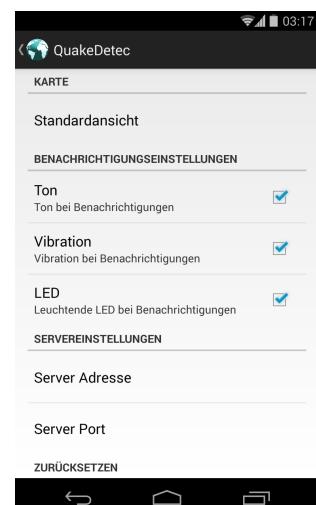


Abbildung 12: Settings

## 6 Erdbebenerkennung unter Android (Christopher Althaus)

Die Erdbebenerkennung innerhalb von Android ist ein wesentlicher Bestandteil des gesamten Systems. Die Realisierung folgt dabei im wesentlichen zwei Grundprinzipien. Zum einen soll die Empfindlichkeit bewusst hoch sein, zum anderen soll der Akkuverbrauch so gering wie möglich gehalten werden.

Die hohe Empfindlichkeit ist darin begründet, dass es besser ist einen Fehlalarm auszulösen, als ein reales Erdbeben nicht als solches zu erkennen. Der möglichst geringe Akkuverbrauch ist der Benutzerakzeptanz der Anwendung zuzuschreiben. Kein Anwender möchte eine Software installieren, welche den Akku in kürzester Zeit leert. Um den Akkuverbrauch gering zu halten, bedarf es somit eines möglichst einfach aufgebautem Erkennungsalgorithmus für Erdbeben.

### 6.1 Abfragen der Sensordaten unter Android

Innerhalb der Android Anwendung werden die Sensordaten in einem Service im Hintergrund abgefragt. Dieser Service nutzt dazu den sogenannten *SensorManager* aus dem *android.hardware* Paket. Mittels des *SensorManagers* ist es möglich, auf alle Sensoren eines Android Gerätes zuzugreifen. Daraufhin wird ein neuer Sensor vom Typ *Accelerometer* angelegt, welcher daraufhin im *SensorManager* registriert werden kann. Im Listing 8 ist der dafür nötige Code aufgeführt.

Listing 8: Abfragen der Sensordaten unter Android

```
1 public final String ACCEL_SAMPLE = "com.th.nuernberg.quakedetec.ACCEL_SAMPLE";
2 public final String ACCEL_SAMPLE_KEY = "ACCELERATION_SAMPLE";
3 SensorManager sensManager = getSystemService(Context.SENSOR_SERVICE);
4 Sensor accel = sensManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
5 sensManager.registerListener(sensListener, accel, SensorManagerSENSOR_DELAY_UI);
6
7 private final SensorEventListener sensListener = new SensorEventListener() {
8     public void onSensorChanged(SensorEvent e) {
9         float x = e.values[0];
10        float y = e.values[1];
11        float z = e.values[2];
12        long t = System.currentTimeMillis();
13        AccelSample accelSample = new AccelSample(x, y, z, t);
14        broadcastSingleSample(accelSample);
15    }
16};
17
18
```

```

19 protected void broadcastSingleSample(AccelSample sample) {
20     Intent intent = new Intent(ACCEL_SAMPLE);
21     intent.putExtra(ACCEL_SAMPLE_KEY, (Parcelable) sample);
22     sendBroadcast(intent);
23 }

```

Zunächst wird, wie bereits geschrieben, der *SensorManager* und die Variable *accel* vom Typ *Sensor* als Beschleunigungssensor initialisiert. Daraufhin wird mittels des *SensorManagers* ein *SensorEventListener* namens *sensListener* für den angelegten Beschleunigungssensor registriert. Dieser *SensorEventListener* wird daraufhin bei jedem Event des Beschleunigungssensor aufgerufen. Beim Registrieren des Sensors wird zudem die Aktualisierungsrate mit angegeben. Diese ist momentan auf den Wert *SENSOR\_DELAY\_UI* eingestellt, was einer eher mittleren Aktualisierungsrate entspricht. Würde die Rate jedoch höher gesetzt werden, würde dies den Akkuverbrauch in die Höhe treiben. Die nächst niedrigere Stufe wäre *SENSOR\_DELAY\_NORMAL*. Sie wird innerhalb von Android zur Displayausrichtung genutzt und hat somit eine zu langsame Aktualisierungsrate, dass sie zur Erkennung von Erdbeben nicht geeignet wäre.

Im *SensorEventListener* werden nun die aktuellen Sensorwerte entgegengenommen. Dabei handelt es sich um die Beschleunigungswerte der X-, Y- und Z-Achse. Die Sensordaten werden nun zusammen mit der aktuellen Systemzeit in einer Variable vom Typ *AccelSample* gespeichert. Diese Klasse dient lediglich dazu, die Sensordaten eines Zeitpunktes innerhalb des Systems mittels eines Broadcasts zu verbreiten. Dieser Broadcast wird von der Methode *broadcastSingleSample* ausgelöst.

Prinzipiell könnte auch innerhalb des *SensorEventListeners* die Auswertung der Beschleunigungssensordaten stattfinden. Da jedoch neben der Erdbebenerkennung auch das Diagramm, welches der Nutzer innerhalb der App sieht, diese Daten benötigt, ist es der übliche Weg, einmal innerhalb einer Applikation die Daten vom Sensor anzufragen und daraufhin innerhalb der Anwendung diese Daten mittels eines Broadcast zu verteilen.

Wie in der Implementierung der *broadcastSingleSample* Methode zu sehen, wird innerhalb dieser Methode ein Intent angelegt. Intents werden vom Android Betriebssystem für den asynchronen Austausch von Nachrichten verwendet. Dem angelegten Intent wird dabei der String *ACCEL\_SAMPLE* als Argument übergeben. Hierbei handelt es sich um eine Art Schlüssel für die auszuführende Aktion des Intents, welcher später dazu benutzt werden kann, Intents dieses Schlüssels abzufangen und zu verwenden. Ebenso werden dem Intent die Beschleunigungssensordaten angehangen. Dies geschieht mittels einer Key/Value Angabe. Dabei ist der String *ACCEL\_SAMPLE\_KEY* der Schlüssel für die übergebenen Beschleunigungssensordaten der Variable *sample*. Mittels des hier angegebenen Schlüssels können Empfänger dieses Intents, die Daten, welche ihm hinzugefügt worden sind, zweifelsfrei zuordnen. Zum Ende der Methode wird das Intent als Broadcast verschickt.

Möchte nun eine Activity oder ein anderer Service auf die Daten des Intents zugreifen, wird ein Broadcast-Receiver benötigt, welcher auf Intents mit dem Schlüssel aus der Variable *AC-*

*ACCEL\_SAMPLE* wartet. Eine beispielhafte Implementierung eines solchen Broadcast-Receivers ist dabei im nachfolgenden Listing 9 dargestellt.

Listing 9: Broadcast-Receiver zum Auslesen der Beschleunigungssensordaten

```

1 private class AccelerationBroadcastReceiver extends BroadcastReceiver {
2
3     @Override
4
5     public void onReceive(Context context, Intent intent) {
6
7         if (intent.getAction().equals(Accelerometer.ACCEL_SAMPLE)) {
8
9             AccelSample sample =
10
11                 intent.getParcelableExtra(Accelerometer.ACCEL_SAMPLE_KEY);
12
13             if (sample != null) {
14
15                 //Erdbebenauswertung
16
17             }
18
19         }
20
21     }
22 }
```

Wie im Quellcodeauszug ersichtlich, empfängt ein Broadcast-Receiver zunächst jegliche auftretende Intents. Jedoch kann unter der Verwendung des Schlüssels, welcher dem Intent zugewiesen worden ist, jedes eingehende Intent dahingehend überprüft werden, ob es sich um das gefragte Intent handelt oder nicht. Ist dies der Fall, so können die dem Intent angehangenen Daten ausgelesen werden. Hierzu wird der Schlüssel verwendet, welcher dem angehangenen Inhalt zugewiesen worden ist. In diesem Fall also der Wert von *ACCEL\_SAMPLE\_KEY*, der Klasse *Accelerometer*. Mit dem ausgelesenen Wert des Intents stehen nun im Broadcast-Receiver die Beschleunigungswerte eines einzelnen Zeitpunktes zur Verfügung. Somit können diese für die Erdbebenauswertung genutzt werden, welche im folgenden erläutert werden soll.

## 6.2 Erdbebenauswertung

Wie bereits erwähnt, soll der zur Auswertung benutzte Algorithmus möglichst ressourcenschonend implementiert werden. Deshalb ist gleich zu Beginn der Implementierung beschlossen worden, dass die Erkennung nicht alle drei Achsen (X, Y, Z) auswerten soll, sondern den Betrag aller Beschleunigungen.

Prinzipiell kann man sich die vom Beschleunigungssensor erhaltenen Werte als Vektoren vorstellen, welche einen rechtwinkligen Raum aufspannen. Möchte man nun den Betragswert dieser Vektoren bestimmen, genügt es, die Raumdiagonale  $r$  dieses Raumes zu berechnen. Diese kann mit der Formel  $|r| = \sqrt{x^2 + y^2 + z^2}$  berechnet werden. Die Formel beruht dabei auf dem Satz des Pythagoras. Zunächst wird dabei die Diagonale der Grundfläche, welche durch zwei Achsen aufgespannt wird, berechnet ( $d = \sqrt{x^2 + y^2}$ ). Daraufhin wird der Satz des Pythagoras ein erneutes Mal auf die ver-

bleibende Achse angewendet ( $|r| = \sqrt{d^2 + z^2}$ ). Durch das Quadrieren von  $d$  fällt dessen Wurzel weg, womit sich die oben genannte Formel ergibt.

Da auf der Erde stets eine Erdanziehungskraft von etwa  $9,81 \frac{m}{s^2}$  wirkt, liegt der berechnete Betragswert der Beschleunigung bei einem still liegendem Gerät stets bei etwa  $9,81 \frac{m}{s^2}$ . Um die Erkennung von Beschleunigungsänderungen zu vereinfachen, ist deshalb entschieden worden, die Erdbeschleunigung vom Betragswert abzuziehen. Auf diese Weise ergibt sich letztendlich die Formel  $|a| = \sqrt{x^2 + y^2 + z^2} - 9,81 \frac{m}{s^2}$  zur Berechnung der betragsmäßigen Beschleunigung des Gerätes. Wird nun das Gerät bewegt, ergeben sich Ausschläge um den Nullpunkt herum, wie in Abbildung 13 dargestellt.



Abbildung 13: Beschleunigungssignal nach der Zusammenfassung

Die dargestellten Ausschläge sind eine Aufzeichnung des Beschleunigungssignals, während das Gerät mit erdbebenartigen Schlägen bewegt worden ist. Durch die Rüttelbewegung des Gerätes wirkt darauf abwechselnd eine Beschleunigung in Richtung der Erdanziehung, wodurch ein positiver Ausschlag entsteht und daraufhin eine Beschleunigung entgegengesetzt der Erdanziehung. Dadurch wird die Erdanziehung aufgehoben und es entsteht ein negativer Ausschlag.

Da bei Erdbeben in kurzer Zeit eine hohe Anzahl von Schlägen auf das Gerät einwirken, ist es in Anbetracht dieser Tatsachen möglich, das Beschleunigungssignal mittels der Zählung von Nulldurchläufen des Signals auf erdbebenartige Schläge hin zu untersuchen.

Durch die Empfindlichkeit des Beschleunigungssensor sind jedoch auch in einer absoluten Ruheposition des Gerätes stets Nulldurchläufe des Signals messbar. Daher ist es notwendig, erst Signalwerte, die über einem gewissen Schwellwert liegen, als Nulldurchläufe zu zählen. Dieser Umstand soll in der folgenden Abbildung 14 verdeutlicht werden.



Abbildung 14: Beschleunigungssignal mit Schwellwerten

Die grünen Linien innerhalb des Diagramms sind dabei die definierten Schwellwerte. Hierzu ist in der Implementierung ein Wert von  $\pm 0,5 \frac{m}{s^2}$  gewählt worden. Wie im Diagramm ersichtlich, stellt

dieser Schwellwert bei wahrnehmbaren Erschütterungen keinerlei Einschränkung dar und filtert zuverlässig kleinere Signalstörungen heraus.

Die eigentliche Erkennung von Erdbeben gestaltet sich nun denkbar einfach und somit auch ressourcenschonend. Für die eigentliche Signalanalysierung werden lediglich zwei Variablen benötigt. Eine Variable legt dabei fest, ob der letzte Signalwert positiv oder negativ war. Die zweite Variable wird bei jedem Signalwechsel von positiv auf negativ erhöht. Betrachtet man diese Auswerte-methode anhand des verwendeten Signalverlaufs der vorherigen Abbildungen, so ergibt sich ein Auswerteergebnis, welches in Abbildung 15 dargestellt ist.

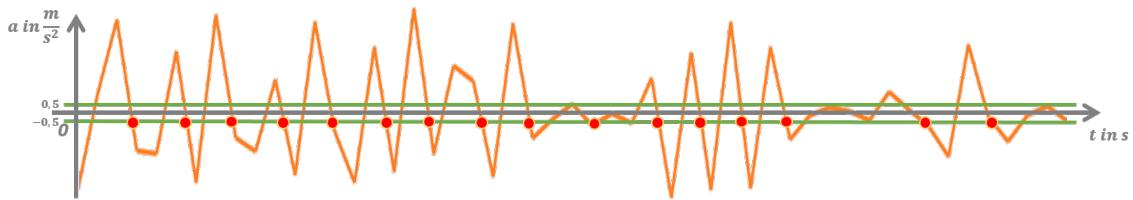


Abbildung 15: Beschleunigungssignal mit Nulldurchläufen analysiert

Jede rote Markierung des Signals steht hierbei für einen Signalwechsel. Wie ersichtlich, ergibt sich somit für jeden einzelnen Ausschlag eine Markierung. Zum Ende des Signals wird dabei auch die Filterung mittels der Schwellwerte deutlich. Hier erreicht ein Signalwechsel diesen Schwellwert nicht und wird deshalb auch nicht mitgezählt.

Zur Erkennung von Erdbeben wird nun in einem bestimmten Zeitrahmen die Anzahl dieser Signalwechsel gezählt. In der momentanen Implementierung wird dazu ein Zeitraum von fünf Sekunden ausgelöst.

Nachfolgend soll die Implementierung in Java erläutert werden. Im Listing 10 sind hierzu die benötigten globalen Variablen aufgeführt.

Listing 10: Globale Variablen der Erbebenerkennung

```

1 private boolean signalGreaterZero = true;
2 private int signalChangeCount = 0;
3 private long analyseTimeSpan = System.currentTimeMillis();
4 private int accelfreq = 0;
```

Die Variable *signalGreaterZero* speichert jeweils für den letzten Signalwert, ob dieser positiv oder negativ war. Die Variable *signalChangeCount* ist für die Zählung der Signalwechsel verantwortlich. In der Variable *analyseTimeSpan* dient zur Speicherung der Systemzeit, um alle fünf Sekunden eine Auswertung zu machen. Die Variable *acceFreq* dient dem Auswerten der Frequenz des Beschleunigungssensor. Dieser Umstand wird nachfolgend noch genauer erläutert.

Anschließend soll die eigentliche Signalauswertung erklärt werden. Dazu ist zunächst, wie bereits beschrieben, die Zählung der Signalwechsel nötig. Die dafür notwendige Implementierung ist im

nachfolgendem Listing 11 aufgeführt.

Listing 11: Implementierung der Erdbebenerkennung

```
1 AccelSample sample = intent.getParcelableExtra(Accelerometer.ACCEL_SAMPLE_KEY);  
2     if (sample != null) {  
3         if (sample.abs < -0.5 && signalGreaterZero) {  
4             signalChangeCount++;  
5             signalGreaterZero = false;  
6         } else if (sample.abs > 0.5)  
7             signalGreaterZero = true;  
8     ...  
9 }
```

Wie im Auszug ersichtlich, ist die Zählung der Signalwechsel mittels des Beschleunigungssensorwertes aus dem Intent des bereits beschriebenen Broadcast-Receiver implementiert. In der Abfrage wird zunächst überprüft, ob der Wert des Sensors größer ist, als der vorgegebene Schwellwert von  $+/- 0.5 \frac{m}{s^2}$ . Ist dies der Fall, wird überprüft ob ein Vorzeichenwechsel des Signals stattgefunden hat. Ist dies der Fall, wird beim Vorzeichenwechsel von positiv zu negativ die Variable *signalChangeCounter* erhöht und das neue Vorzeichen mittels der Variable *signalGreaterZero* gespeichert. Nachdem nun die Nulldurchläufe gezählt werden, wird eine Auswertung der gezählten Durchläufe in einem bestimmten Zeitrahmen benötigt. Diese wird im nachfolgenden Listing 12 erläutert.

Listing 12: Analyse der gezählten Nulldurchläufe

```
1 if (System.currentTimeMillis() - analyseTimeSpan > 5000) {  
2     double alarmRatio = 0;  
3     if(signalChangeCount != 0)  
4         alarmRatio = (double)signalChangeCount/(double)accelFreq * 100.0;  
5         //Alarmauswertung abhängig von der Frequenz  
6         double alarmFrequRel = accelFreq * 0.01 + 0.5;  
7         alarmRatio = alarmRatio * alarmFrequRel;  
8  
9     if (alarmRatio > 25)  
10         sendAlarmToServer();  
11  
12     analyseTimeSpan = System.currentTimeMillis();  
13     accelFreq = 0;  
14     signalChangeCount = 0;  
15 }  
16 accelFreq++;
```

Die Analyse der gezählten Signalwechsel wird dabei alle fünf Sekunden ausgeführt. Dazu wird die Variable *analyseTimeSpan*, welche den Zeitpunkt der letzten Analyse abspeichert, bei jedem Aufruf des Broadcast-Receivevers mit der aktuellen Systemzeit verglichen. Nachdem eine Zeitspanne von über fünf Sekunden vergangen ist, wird die Analyse gestartet.

Wie ersichtlich, reicht es nicht aus, lediglich die Anzahl der gezählten Spitzen zur Auswertung heranzuziehen. Das liegt daran, dass der Sensor innerhalb von Android keine gleichbleibende Frequenz besitzt und sich zudem die Frequenz von Gerät zu Gerät massiv unterscheiden kann. Die Frequenz der Sensoren reicht dabei von 2 Hz bis hin zu 50 Hz. Da bei einer Frequenz von 2 Hz innerhalb von fünf Sekunden lediglich 10 Messwerte verfügbar sind, ist hier eine zuverlässige Erkennung kaum möglich. Für die Auswertung wird deshalb eine Frequenz von 10 Hz als optimal angesehen und Frequenzen, die darüber oder darunter liegen dementsprechend mittels der Berechnung der Variable *alarmFrequRel* angepasst. Die Auswahl der Frequenz von 10 Hz als Referenz begründet sich vor allem dadurch, dass sich während der Implementierung herausgestellt hat, dass die meisten Geräte den Beschleunigungssensor durchschnittlich mit dieser Frequenz aktualisieren. Anhand dieser Referenzfrequenz ist definiert worden, dass ab einem Verhältnis der gemessenen Signalwechsel zu den gesamten gemessenen Werten von größer 25% ein Alarm ausgelöst werden soll. Da bei einer höheren Frequenz die Messwerte eines einzelnen Signalwechsels deutlich zunehmen, muss hier die Gewichtung von gemessenen Signalwechseln zu gemessenen Messwerten erhöht werden. Ist die Frequenz langsamer als 10 Hz, wird die Gewichtung der Signalwechsel dementsprechend reduziert. Ist die Variable *alarmRatio* größer als 25, wird mittels der Methode *sendAlarmToServer* ein Alarm an den Server geschickt. Nach der Auswertung werden allezählenden Variablen wieder zurückgesetzt und der Zeitpunkt der Analyse in der Variable *analyseTimeSpan* vermerkt, um nach fünf Sekunden erneut eine Analyse starten zu können.

Insgesamt sind die Parameter der Analyse während der Testphase stets angepasst worden und lösen nun bei allen zur Verfügung stehenden Testgeräten mit einer annähernd gleichen Empfindlichkeit einen Alarm aus.

## 7 Lokalisierung (Niklas Schäfer)

### 7.1 Einleitung

Essentiell für die Auswertung eines Erdbebens mit Smartphones ist die Lokalisierung. Empfängt der Server Alarne von den Geräten, ist es notwendig, dass er unter anderem auch die Standortdaten von der App geliefert bekommt. Zum Einen benötigt man eine Lokalisierung um feststellen zu können, wo sich das Beben befindet und zum Anderen muss der Server anhand der Standorte der empfangenen Alarne auswerten können, ob es sich um einen korrekten Alarm oder einen Fehlalarm handelt. Zum Beispiel ist es möglich, dass das Fahren in einem Bus den Beschleunigungssensor Bewegungen wahrnehmen lässt, die einem Erdbeben ähnlich sind, sodass die App von einem Erdbeben ausgehen muss und einen Alarm an den Server sendet. Da der Server bei Alarmen basierend auf Mehrheitsentscheidung urteilt, könnte es passieren, dass er auf ein Erdbeben schließt, wenn mehrere Personen in einem Bus fahren und Alarne senden. Daher muss gewährleistet werden, dass es mindestens einen Alarm gibt, der einen bestimmten Mindestabstand zu den betreffenden anderen Alarmen aufweist.

### 7.2 Location Provider des Android-Systems

Im Android System gibt es drei unterschiedliche Location Provider (Verfahren und Sensoren, die einen Standort bestimmen können), die Positionsdaten eines Android Smartphones liefern können:

- PASSIVE
- NETWORK
- GPS

#### 7.2.1 PASSIVE

Der Location Provider *PASSIVE* initialisiert keinen Zugriff auf einen der vorhandenen Sensoren/Ortungsdienste des Smartphones. Er nutzt lediglich den zuletzt abgerufenen Standort, der sich noch im Speicher des Android Systems befindet. Das heisst, dass er nur den Standort liefert, der von einer beliebigen anderen App oder dem Android System selbst zu irgendeinem Zeitpunkt über die Location Provider *NETWORK* oder *GPS* abgerufen worden ist. Der Vorteil des *PASSIVE* Providers liegt darin, dass er keinen Sensor abfragen muss, sondern lediglich den Speicher ausliest und daher sehr stromsparend fungiert. Für die Quakedetec App ist er allerdings nicht sinnvoll verwendbar, da mit sehr hoher Wahrscheinlichkeit kein ausreichend aktueller Standort vorhanden ist. Es ist beispielsweise möglich, dass die letzte Standortabfrage schon sehr lange zurück liegt und der Standort somit veraltet ist. Somit wäre eine Erdbebenlokalisierung fehlerhaft.

### 7.2.2 NETWORK

Der Location Provider *NETWORK* findet Positionsdaten mit Hilfe von WLAN Netzwerken und der Mobilfunkzellen ("Mobilfunkmast"). Um über WLAN Netzwerke Geräte lokalisieren zu können, pflegt Google eine Datenbank, die WLAN Netzwerke mit Standortdaten verknüpft. Anfangs geschah dies mit Hilfe des Google Cars, welches hauptsächlich zur Datenerfassung für Google StreetView in vielen Städten unterwegs war. Hierbei erfasste dieses Fahrzeug neben den Straßenbildern auch WLAN Netze und verknüpfte deren MAC-Adresse mit den GPS Daten des Fahrzeugs. Natürlich sind solche Daten schnell veraltet, weshalb Google weiterhin Android Smartphones nutzt, um die Daten aktuell zu halten und neue WLAN Netze in die Datenbank aufzunehmen. Android Smartphones übermitteln dabei ihren *GPS* Standort, sobald ein solcher ermittelt wurde, zusammen mit den MAC IDs der sichtbaren WLAN Netzwerke in ihrer Umgebung an Google. Dabei wird auch die Signalstärke der WLAN Netzwerke berücksichtigt, um den Standort des WLAN Netzwerks noch genauer ermitteln zu können. Um einen Standort über den *NETWORK* Provider abzurufen, sendet das Smartphone einen *Request* an Google. In diesem *Request* sind ebenfalls die MAC IDs und Signalstärken der gerade sichtbaren WLAN Netze enthalten. Auch hier werden diese Daten und nicht nur das verbundene WLAN benötigt, um den Standort weiter präzisieren zu können. Würde man nur das verbundene WLAN übermitteln, könnte nur der Standort des einzelnen WLAN Netzes zur Ermittlung des Standorts verwendet werden. Hat dieses WLAN eine hohe Signalstärke, sodass es in einem Radius von mehreren hundert Metern empfangbar ist, wäre der Standort auch nur auf mehrere hundert Meter genau, da nicht bestimmt werden kann, wo sich das Gerät in diesem Empfangsbereich befindet. Bezieht man aber mehrere WLAN Netzwerke mit ein, kann man durch die Überschneidungen der Netze den Standort wesentlich genauer bestimmen.

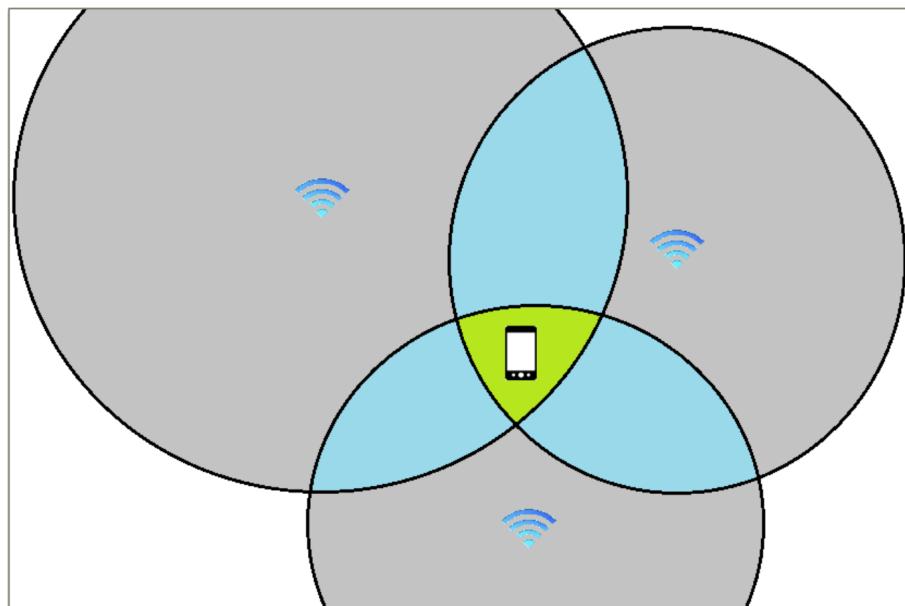


Abbildung 16: Lokalisierung mit *NETWORK* Provider

In Abbildung 16 ist die Funktionsweise dieser Lokalisierungstechnik skizzenhaft dargestellt. Ist am Standort allerdings nur ein WLAN Netzwerk sichtbar/verfügbar, bezieht auch hier Google die Signalstärke mit ein, wodurch auch in diesem Fall die Genauigkeit weiter verbessert werden kann. Wurde der Request vom Smartphone an den Google Server übergeben, werden anhand der gesendeten MAC IDs die verfügbaren Standorte der umgebenden WLAN Netze aus der Google Datenbank entnommen. Daraufhin wird ein Standort mit der beschriebenen Funktionsweise ermittelt (Überschneidungen) und an das Smartphone zurückgesendet. Die Antwort vom Google Server beinhaltet den Längen- und Breitengrad, die Genauigkeit des gelieferten Standorts und einen Zeitstempel. Die Genauigkeit dieses Verfahrens liegt erfahrungsgemäß bei 15-100m.

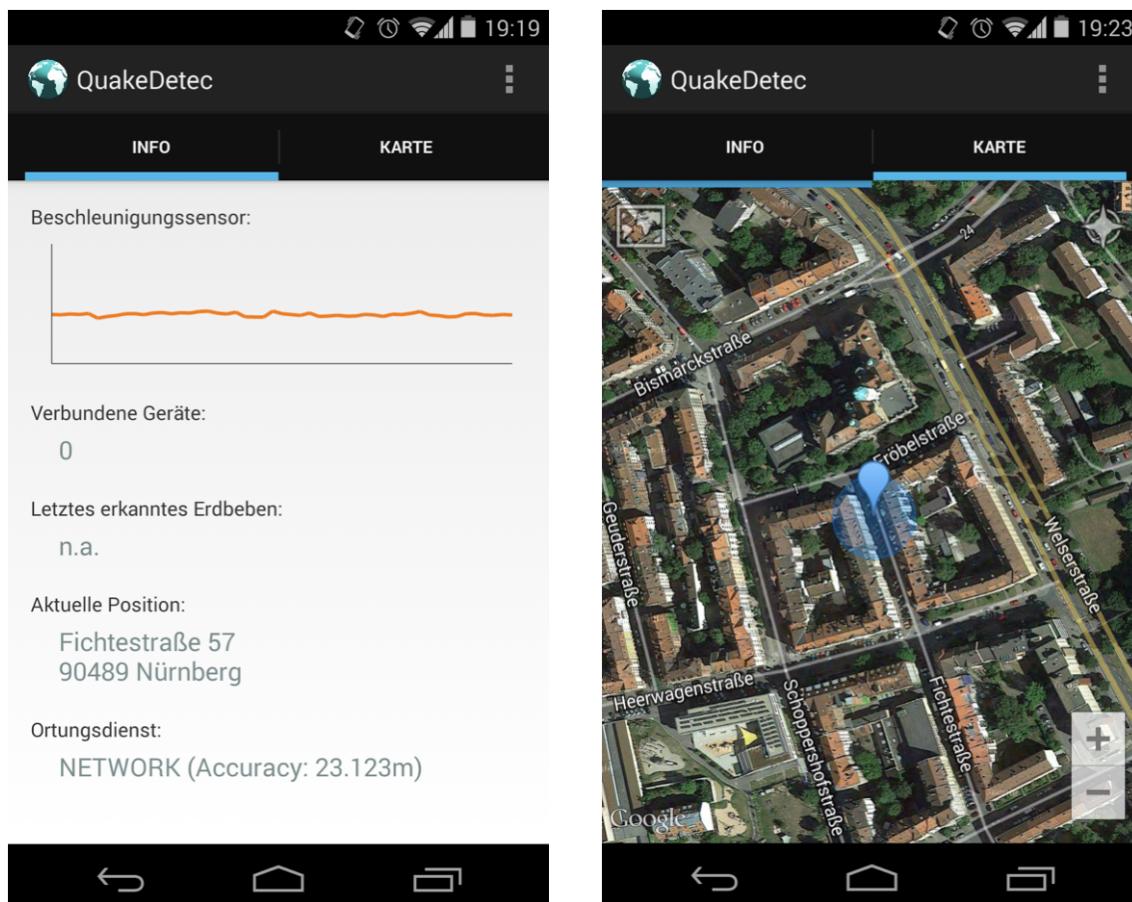


Abbildung 17: User Interface bei Nutzung des NETWORK Providers

Sollten keine WLAN Netzwerke verfügbar oder das WLAN des Smartphones deaktiviert sein, nutzt der *NETWORK* Provider die Mobilfunkzellen zur Ortsbestimmung. Dies funktioniert im Grunde nach einem ähnlichen Prinzip, wie die Lokalisierung über WLAN Netze, allerdings werden hierbei keine Überschneidungen mehrerer Mobilfunkzellen berücksichtigt. Google ist der Standort des Mobilfunkmasts ebenfalls bekannt. Um aber auch bei diesem Verfahren mit Überschneidungen arbeiten zu können, müsste das sogenannte Timing-Advance Verfahren verwendet werden. Bei diesem Verfahren wird der Standort ermittelt, indem Signallaufzeiten zum verbundenen Mobilfunkmast

gemessen werden. Wird nur die Signallaufzeit zu einem Mast berechnet, kann nur bestimmt werden, welchen Abstand das Gerät zum Mobilfunkmast hat. Um den Standort mit Hilfe von Signallaufzeiten genauer zu bestimmen, ist es notwendig die Signallaufzeit zu mindestens einem weiteren Mobilfunkmast zu messen.

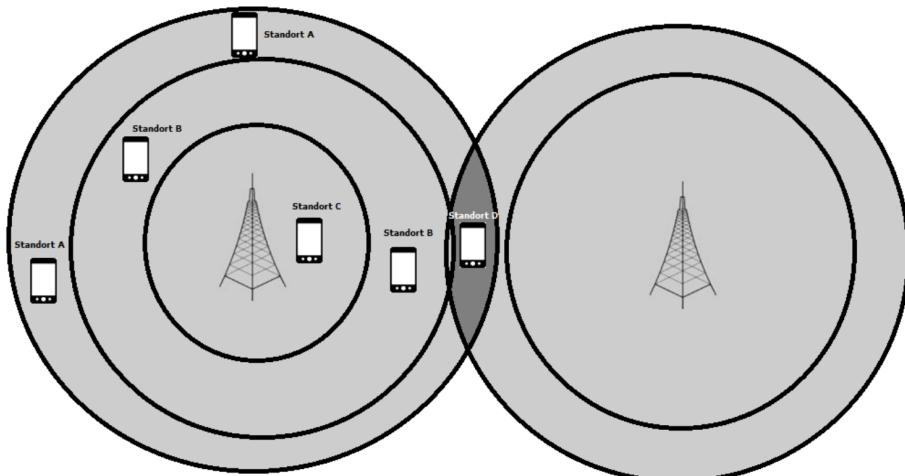


Abbildung 18: Timing-Advance Verfahren

Somit gibt es zwei Räume, die sich schneiden und es kann relativ genau ein Standort ermittelt werden. Dieses Verfahren findet allerdings kaum Verwendung, da das Gerät dazu gezwungen werden muss die Verbindung zwischen mehreren Mobilfunkzellen zu wechseln, um in jeder Zelle die Signallaufzeit zu messen. Dieser Vorgang führt schnell zu einem hohen Energieverbrauch. Somit wird in der Praxis eine Lokalisierung nur mit Hilfe einer einzigen Mobilfunkzelle durchgeführt.

Eine Mobilfunkzelle hat eine vielfach höhere Signalreichweite als ein WLAN Netzwerk. In Städten liegt man hier zwar nur bei wenigen hundert Metern, auf dem Land hingegen bei mehreren Kilometern. Somit erhält man in der Stadt meist noch Standortdaten, die auf wenige hundert Meter genau sind, auf dem Land liegt man allerdings meist bei 2 bis 4 Kilometer.

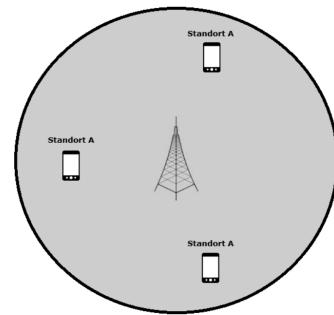


Abbildung 19: Praxis

### 7.2.3 Global Positioning System

Mittlerweile besitzt fast jedes aktuelle Android Smartphone einen *GPS* Empfänger. Das *Global Positioning System* wurde vom amerikanischen Verteidigungsministerium entwickelt. Es besteht aus 30 Satelliten, welche die Erde umkreisen und Nachrichten aussenden, die zur Positionsbestimmung eines Empfängers dienen. Das *Global Positioning System* ist vom Prinzip her der Positionsbestimmung über Mobilfunkzellen im Timing-Advance Modus ähnlich, allerdings im dreidimensionalen Raum. Ein Satellit sendet eine Nachricht mit seiner Position und einem Zeitstempel.

pel kontinuierlich aus. Ein *GPS* Empfänger kann aus der Versandzeit einer Nachricht die Signallaufzeit bestimmen (aktuelle Zeit - Versandzeit = Signallaufzeit). Aufgrund der Signallaufzeit kann somit bestimmt werden, in welchem Abstand sich der Empfänger zum Satelliten befindet. Ein einziger Satellit würde zu einer genauen Positionsbestimmung nicht ausreichen, da somit nur die Entfernung zu diesem einen Satelliten bekannt ist. Werden aber mehrere Satelliten genutzt, können die Überschneidungen der Radien verwendet werden, um die Position sehr genau zu bestimmen. Mit zwei Satelliten kann theoretisch in einer zweidimensionalen Welt schon eine Position genau bestimmt werden. In Abbildung 20 ist ein Beispiel mit zwei Satelliten zu sehen. Bei der Verwendung von zwei Satelliten gibt es zwei Radien, die sich in zwei Punkten schneiden. Da aber einer dieser Punkte weit im Weltall liegt, kann schnell daraus geschlossen werden, dass nur der andere Punkt Relevant und somit der gesuchte Standort ist. In der Realität wird allerdings noch ein dritter Satellit benötigt, um das sogenannte *Uhrenproblem* zu lösen. Die Satellitenuhren laufen zwar dank Atomuhren absolut genau und synchron, ein *GPS* Empfänger tut dies allerdings nicht. Dadurch würden die Signallaufzeiten bei zwei Satelliten ungenau berechnet werden.

Mit der Hilfe eines dritten Satellitens kann dieses Problem gelöst werden.

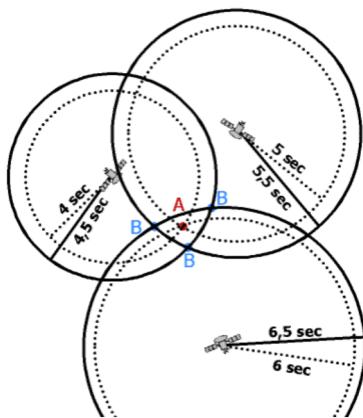


Abbildung 21: GPS 3 Satellites

Abbildung 21 verdeutlicht die Funktionsweise. Unter der Annahme, dass die Empfängeruhr 0,5s vorgeht, bekommt man Radien die aufgrund der falsch laufenden Empfängeruhr größer ausfallen (durchgezogene Linien/Kreise). In Wirklichkeit fallen diese aber kleiner aus (gepunktete Linien/Kreise). Bei zwei Radien sind bei diesem Szenario zwei Schnittpunkte vorhanden, die eine falsche Position angeben würden, ohne dass es eine Möglichkeit gäbe den Fehler zu erkennen. Wird allerdings ein dritter Satellit zur Bestimmung herangezogen, gibt es bei einer falschen Berechnung aufgrund des Uhrenproblems keinen Schnittpunkt aller drei Radien. In diesem Fall verändert der *GPS* Empfänger seine Uhrzeit so lange, bis sich alle drei Radien in einem Punkt schneiden. Somit wird die Uhrzeit des *GPS* Empfängers mit der Uhrzeit der Satelliten synchronisiert. Der Schnittpunkt aller drei Radien ist dann die exakte Position. Zur Realisierung dieses Systems und zur genauen Bestimmung der Position in der Realität, also in einer dreidimensionalen Welt, wird noch ein vierter Satellit verwendet. Dies ist notwendig, da bei der hier beschriebenen Funktionsweise von einem zweidimensionalen Modell ausgegangen wird. Es kann zwar auch mit diesem Modell die

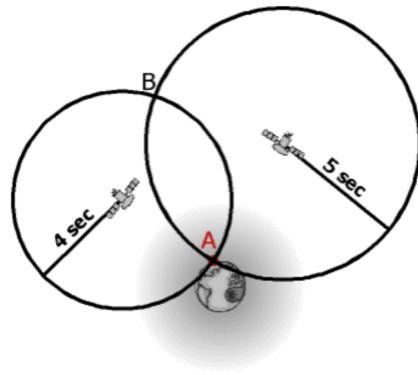


Abbildung 20: GPS 2 Satelliten

Abbildung 21 verdeutlicht die Funktionsweise. Unter der Annahme, dass die Empfängeruhr 0,5s vorgeht, bekommt man Radien die aufgrund der falsch laufenden Empfängeruhr größer ausfallen (durchgezogene Linien/Kreise). In Wirklichkeit fallen diese aber kleiner aus (gepunktete Linien/Kreise). Bei zwei Radien sind bei diesem Szenario zwei Schnittpunkte vorhanden, die eine falsche Position angeben würden, ohne dass es eine Möglichkeit gäbe den Fehler zu erkennen. Wird allerdings ein dritter Satellit zur Bestimmung herangezogen, gibt es bei einer falschen Berechnung aufgrund des Uhrenproblems keinen Schnittpunkt aller drei Radien. In diesem Fall verändert der *GPS* Empfänger seine Uhrzeit so lange, bis sich alle drei Radien in einem Punkt schneiden. Somit

Position in der Realität bestimmt werden, allerdings wird dabei davon ausgegangen, dass sich die zu bestimmende Position auf Meereshöhe befindet. Somit wäre eine Positionsbestimmung falsch, sobald sich der Empfänger über (bzw. unter) dem Meeresspiegel befindet. Bei der dreidimensionalen Positionsbestimmung wird dieses Problem mit einem vierten Satelliten gelöst. Mit diesem ist es auch möglich, die Höhe des Empfängers zu bestimmen. Allerdings soll hier nicht weiter darauf eingangen werden. Dazu der Verweis auf weitere Informationsquellen.

### 7.3 Positionsbestimmung in Android

Wie schon in Kapitel 7.2 beschrieben, bietet das Android System drei Möglichkeiten um den Standort zu bestimmen. Um unter Android diese Möglichkeiten zu nutzen, bietet die Android API die drei Location Provider *PASSIVE*, *NETWORK* und *GPS*. Um einer App den Zugriff auf diese Provider zu gewähren, muss der Zugriff erstmal beim Nutzer erfragt werden. Um diese Berechtigung vor der Installation beim Nutzer einzuholen, muss die folgende Codezeile im Android-Manifest hinzugefügt werden:

Listing 13: App Permissions

```

1 <manifest ... >
2     <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
3     ...
4 </manifest>
```

Möchte ein Nutzer die App nun installieren, wird er zuerst gefragt, ob er der App den Zugriff auf die Standortdaten seines Geräts gewähren möchte. Es ist zwingend erforderlich, dass die Abfrage akzeptiert wird, ansonsten wird die App nicht installiert.

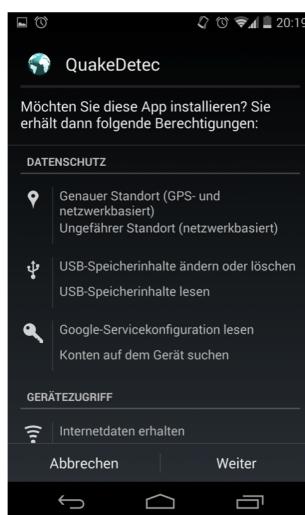


Abbildung 22: App Permissions

Um in der App auf die Location Provider zugreifen zu können, wird eine Instanz des Android *LocationManagers* benötigt. Mit diesem ist es möglich auf die Location Services zuzugreifen und eine Standortbestimmung auszulösen.

Eine Instanz des LocationManagers erlangt man durch folgende Codezeile:

Listing 14: LocationManager Instanz

```
1 // Acquire a reference to the system Location Manager  
2 LocationManager locationManager = (LocationManager) this.getSystemService(Context.LOCATION_SERVICE);
```

Außerdem ist ein LocationListener notwendig, welcher beschreibt, wie auf bestimmte Ereignisse der Location Services reagiert werden soll.

Es gibt vier Ereignisse auf die reagiert werden kann:

- Neuer Standort
- Provider aktiviert
- Provider deaktiviert
- Status der Provider hat sich geändert

Die entsprechenden Methoden des Listeners sind:

- *onLocationChanged(Location location)*
- *onProviderEnabled(String provider)*
- *onProviderDisabled(String provider)*
- *onStatusChanged(String provider, int status, Bundle extras)*

Hat beispielsweise ein Location Provider einen neuen Standort ermittelt, wird die *onLocationChanged* Methode aufgerufen, der ein *Location* Objekt übergeben wird. In diesem Objekt ist der Längen- und Breitengrad, die Genauigkeit des Standortes in Metern, der Location Provider, der den Standort ermittelte, und ein Zeitstempel enthalten.

Wird ein Location Provider (z.B. *GPS*) im System deaktiviert, wird die *onProviderDisabled* (bzw. wenn aktiviert *onProviderEnabled*) Methode mit dem Providernamen als Parameter aufgerufen und man kann dementsprechend in der App darauf reagieren.

Das vierte Ereignis auf das reagiert werden kann, ist eine Statusänderung eines Location Providers. Wenn ein Provider aus irgendeinem Grund nicht verfügbar war und wieder verfügbar wird, wird *onStatusChanged* aufgerufen.

Listing 15: LocationListener

```

1 // Define a listener that responds to location updates
2 LocationListener locationListener = new LocationListener() {
3     public void onLocationChanged(Location location) {
4         // Called when a new location is found by the network location provider.
5         makeUseOfNewLocation(location);
6     }
7     public void onStatusChanged(String provider, int status, Bundle extras) {}
8     public void onProviderEnabled(String provider) {}
9     public void onProviderDisabled(String provider) {}
10 };

```

Um Standortdaten abzufragen, gibt es zwei Möglichkeiten. Eine besteht aus der kontinuierlichen Reaktion auf Standortänderungen. Das heißt, dass Standortänderungen zu jedem Zeitpunkt wahrgenommen werden.

Listing 16: requestLocationUpdates

```

1 locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER, time, distance,
                                         locationListener);

```

Dieser Methode wird unter anderem der *Location Provider* und der entsprechende *LocationListener* übergeben. Ein weiterer Parameter steht für die Zeit, die vergangen sein muss, bis wieder ein neuer Standort zurückgegeben wird. Gibt man zum Beispiel 10 Minuten an, wird für 10 Minuten kein neuer Standort zurückgegeben. Der letzte verbleibende Parameter steht für die Distanz, die überbrückt worden sein muss, bis ein neuer Standort zurückgegeben wird. Setzt man diesen Parameter zum Beispiel auf 100m, wird ein neu erkannter Standort nur zurückgegeben, wenn die Entfernung zwischen diesem und dem alten mindestens 100m beträgt.

Eine andere Möglichkeit einen Standort zu ermitteln, ist ein einmaliger Abruf des aktuellen Standorts.

Listing 17: requestSingleUpdate

```

1 locationManager.requestSingleUpdate(LocationManager.NETWORK_PROVIDER, locationListener,
                                      looper);

```

Dieser Methode wird ebenfalls der *LocationListener* und der gewünschte *Location Provider* übergeben. Zusätzlich wird bei dieser Methode noch ein Looper Objekt benötigt, welches eine Messageschleife

für Threads bereitstellt. Diese Methode aktiviert den gewünschten *Location Provider* und wartet, bis ein Standort von diesem zurückgegeben wurde. Danach wird der *Location Provider* wieder deaktiviert und es werden keine neuen Standorte mehr ermittelt.

## 7.4 Umsetzung der Quakedetec App

### 7.4.1 Erste Umsetzung und Probleme

Anfangs wurde die App mit der *requestLocationUpdates* Methode (siehe Kapitel 7.3) und dem *NETWORK* und *GPS* Provider umgesetzt. Es wurden beide Sensoren kontinuierlich abgefragt und der genauere und aktuellere Standort der beiden Location Provider wurde verwendet. Da man im Falle eines Erdbebens den Standort des Geräts benötigt um die Erdbebenposition zu bestimmen, erachteten wir es als sinnvoll, den Standort kontinuierlich zu bestimmen. Diese Möglichkeit bot die *requestLocationUpdates* Methode. Allerdings fiel schnell auf, dass der Akkuverbrauch mit dieser Lösung sehr hoch und somit inakzeptabel war. Um dieses Problem zu lösen, fügten wir in der App die Einstellung für ein Aktualisungsintervall hinzu. Über diese Einstellung stellten wir eine Möglichkeit zur Verfügung, in der *requestLocationUpdates* Methode den Parameter für die Zeit zu setzen (in Kapitel 7.3 beschrieben). Das bot dem Nutzer die Möglichkeit das Zeitintervall zu bestimmen, welches vergangen sein muss bis ein neuer Standort ermittelt wird. Zum Zeitpunkt der Implementierung war uns nicht bewusst, dass diese Umsetzung keine Auswirkung auf den hohen Akkuverbrauch haben wird. Denn nicht wie zuerst angenommen, sinkt zwischen dem Intervall der Energieverbrauch der Location Provider, sondern er bleibt identisch, da die Location Provider trotz des Zeitintervalls durchgehend aktiv bleiben. Das heißt, dass der Standort trotz des Intervalls durchgehend ermittelt wird, nur dass der Standort ausschließlich nach Verstreichen des Intervalls zurückgegeben wird. Als uns dies bewusst wurde, stellten wir die Lokalisierung auf Singleupdates um (*requestSingleUpdate*, siehe Kapitel 7.3). Diese Singleupdates wurden mit Hilfe eines Timer Threads, welcher in einem Hintergrundservice lief, ausgeführt. Nun wurde das Interval dieses Timer Threads über die Einstellung in der App geregelt. Somit war dies nun ein reales Intervall in dem die Location Provider deaktiviert werden. Das brachte zwar eine Verbesserung der Akkuleistung, aber der Verbrauch war selbst bei einem 10 Minuten Intervall noch zu groß. Außerdem war diese Umsetzung nicht vereinbar mit der benötigten Aktualität der Standortdaten. Denn letztendlich kann sich ein Nutzer innerhalb von 10 Minuten schon an einem ganz anderen Standort befinden. Wurde beispielsweise eine Standortaktualisierung 9 Minuten vor einem Erdbebenalarm ausgelöst und der Benutzer war in dieser Zeit in Bewegung, sind die an den

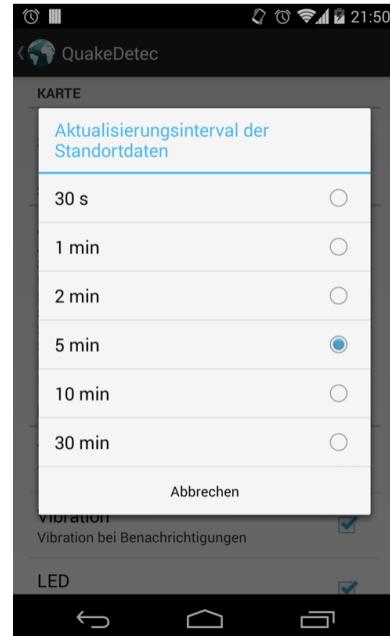


Abbildung 23: Update Interval

Server übermittelten Standortdaten nicht mehr aktuell und können stark von der tatsächlichen Position abweichen. Somit war diese Lösung ebenfalls inakzeptabel und die Implementierung musste überdacht werden.

#### 7.4.2 Finale Umsetzung

Das Problem des hohen Akkuverbrauchs konnte schnell ausfindig gemacht werden. *GPS* ist zwar mit einer Genauigkeit von drei Metern sehr präzise, aber der *GPS* Sensor hat einen sehr hohen Akkuverbrauch. Außerdem besteht ein weiteres Problem von *GPS* darin, dass eine Sichtverbindung zu Satelliten bestehen muss, was zur Folge hat, dass eine Positionsbestimmung innerhalb von Gebäuden nicht möglich ist. Und schließlich werden sich viele Nutzer überwiegend innerhalb von Gebäuden aufhalten. Somit entschlossen wir uns, überwiegend auf *GPS* zu verzichten und primär auf den *NETWORK* Provider zurückzugreifen. Dieser bietet den Vorteil, dass er sehr stromsparend arbeitet und innerhalb von Gebäuden ebenfalls nutzbar ist. In Städten besitzt dieser Provider durch die hohe WLAN- und Funkzellendichte eine sehr hohe Genauigkeit. Bei Tests in Nürnberg lag die Genauigkeit zwischen 20-50m. Leider hat er den Nachteil, dass er auf dem Land sehr ungenau werden kann, da hier oft keine WLAN Netze und nur sehr wenige Mobilfunkzellen vorhanden sind. Bei durchgeföhrten Tests beispielsweise in einem ländlichen Raum in der Nähe Bad Hersfelds und auf Autobahnen, lagen die schlechtesten Werte bei 3400m. Laut Informationen des Geoforschungszentrums Potsdam, die wir durch E-Mail Kontakt erhielten, ist dieser Wert kein sehr großes Problem bei der Erdbebenerkennung, allerdings kommt hierbei zum tragen, dass zum Teil mehrere Benutzer in einem Senderadius eines Mobilfunkmasts genau den gleichen Standort zugeordnet bekommen können, und zwar den des Funkmasts. Das führt zu der Problematik, dass wir bei einer Erdbebenauswertung auf dem Server nicht feststellen können, ob sich die Nutzer genau am selben Standort oder weit voneinander entfernt befinden. Bei der Erdbebenauswertung mit Hilfe einer Mehrheitsentscheidung muss allerdings ausgeschlossen werden, dass sich die Nutzer nicht am gleichen Ort (z.B. in einem Radius von 100m) befinden. Denn sollten z.B. mehrere Nutzer in einem Bus unterwegs sein und aufgrund erdbebenähnlicher Bewegungen des Busses einen Erdbebenalarm an den Server senden, würde bei der Auswertung ein Erdbeben festgestellt werden. Um diesen Fall auszuschließen, wird bei der Erdbebenauswertung immer überprüft, ob sich mindestens ein Gerät der alarmauslösenden Geräte mindestens 100m von den anderen entfernt be-

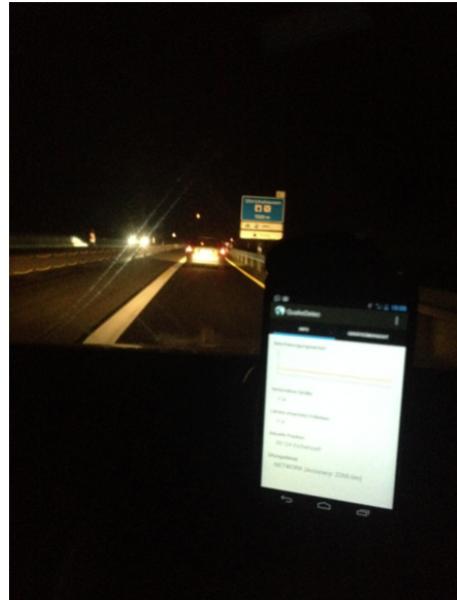


Abbildung 24: Test auf der Autobahn

findet. Das kann im schlechtesten Fall im ländlichen Raum bedeuten, dass ein Gerät mindestens soweit von den anderen entfernt sein muss, sodass es mit einer anderen Mobilfunkzelle verbunden ist. Das heißt bei dem von uns am schlechtest gemessenen Wert von 3400m, muss eines von den alarmauslösenden Geräten mindestens 3401m vom Funkmast entfernt sein, damit es einen anderen Standort zugewiesen bekommt und man sich bei der Auswertung sicher sein kann, dass sich die Geräte nicht am selben Standort befinden. Um zu gewährleisten, dass die Ungenauigkeit nicht zu groß wird, wurde die Lokalisierung so implementiert, dass bei einer Genauigkeit höher als 3500m der GPS Sensor aktiviert wird und über diesen ein genauer Standort abgerufen wird. Das heißt zusammenfassend, dass die App im seltenen worst-case Szenario nur Erdbeben erkennt, die weiter strahlen als 3500m, sodass Geräte in einem Radius, der größer als 3500m ist, das Beben noch wahrnehmen können. Die Implementierung auf diese Weise stellte den besten Kompromiss zwischen Akkuverbrauch und Genauigkeit dar, da der GPS Sensor somit nur sehr selten verwendet wird. Ebenfalls wurde Rücksprache mit dem Geoforschungszentrum Potsdam gehalten, wo uns mitgeteilt wurde, dass Erdbeben, die weniger als 3500m strahlen, so schwach sind, dass von ihnen keine Gefahr ausgeht.

”Ihre Frage war, ob für ein Beben, das nur in einem Umkreis von 3,5km gespürt werden kann, eine Frühwarnung ausgesprochen werden sollte. Unserer Meinung nach kann man darauf verzichten, denn es kann sich aufgrund des eingeschränkten Bereiches (gefühlt in kleiner 3,5km) nur um ein kleines Beben handeln, das kein Schadenspotenzial besitzt.”  
(Claus Milkereit, GFZ Potsdam)

Somit ist es unproblematisch auf eine Erdbebenerkennung zu verzichten, die auch Erdbeben mit einem Wirkungsradius von unter 3500 Metern erkennt.

Da die App abhängig von den Standortdaten ist und ohne Lokalisierung nutzlos ist, prüft die App, ob zumindest eine der beiden Lokalisierungsmethoden (*NETWORK* oder *GPS*) im Android System aktiviert ist. Ist dies nicht der Fall, wird bei geöffneter App ein Popup angezeigt, welches den Nutzer auffordert, die Lokalisierung zu aktivieren. Tippt der Nutzer auf "Öffnen", werden die Android Systemeinstellungen für die Lokalisierung geöffnet. In der App schließt sich das Popup erst, wenn eine der beiden Lokalisierungsmethoden aktiviert wurde.

Läuft die App im Hintergrund und der Nutzer deaktiviert die Lokalisierung, gibt die App eine Android Notification in der Android Statusbar aus. Diese kann vom Nutzer nicht geschlossen werden und schließt sich ebenfalls erst nach der Aktivierung einer Lokalisierungsmethode und kann nicht vom Nutzer geschlossen werden.

Um das Problem der Aktualität der Daten zu lösen, wurde die Standortabfrage flexibel gestaltet. Ist die App geöffnet und sichtbar, wird die Standortabfrage in einem Intervall von 30s ausgeführt. Das Intervall ist bei geöffneter App kurz gewählt, damit gewährleistet wird, dass der Nutzer in der Standortanzeige und der Google Map einen relativ aktuellen Standort angezeigt bekommt. Ist die App nicht sichtbar und läuft im Hintergrund, wird eine Standortabfrage in einem Intervall von 10 Minuten ausgeführt. Diese Abfrage ist notwendig, da die App alle 10 Minuten einen Heartbeat mit dem Standort an den Server sendet. Außerdem wird ein Standort abgefragt, sobald ein Erdbeben erkannt wurde und wird mit dem Alarm an den Server gesendet. Da ein Alarm in der App zeitkritisch ist und eine Standortabfrage mit Hilfe des GPS Providers eine längere Zeit beanspruchen kann, wird zusätzlich in einem Intervall von fünf Minuten eine Standortabfrage unabhängig vom Heartbeat oder Alarm ausgeführt. Somit kann sichergestellt werden, dass ein Standort der mit einem Alarm gesendet wird, nicht älter als fünf Minuten ist, falls die Standortabfrage unmittelbar vor dem Alarm zuviel Zeit in Anspruch nimmt und nicht auf diese gewartet werden kann.

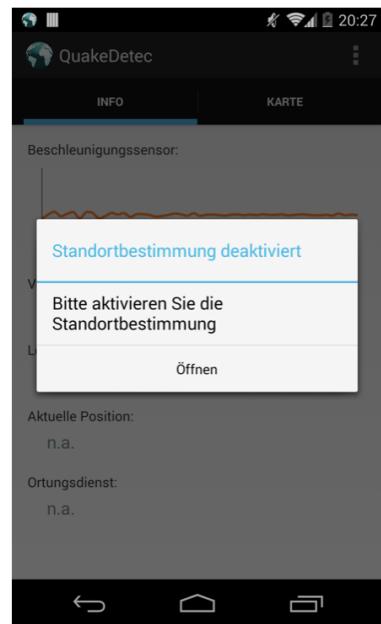


Abbildung 25: Notifications

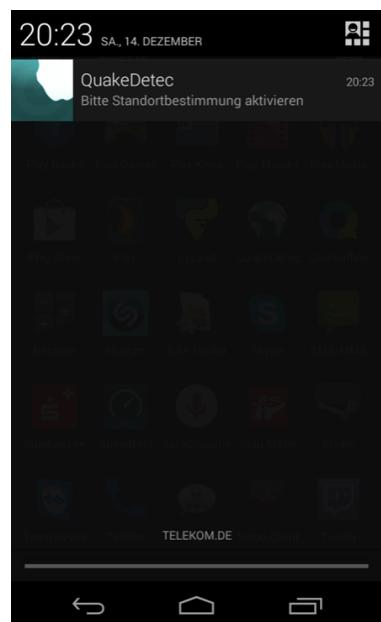


Abbildung 26: Notifications

Zusammenfassung der Lokalisierungsintervalle:

Auslöser	Intervall
Unabhängige Lokalisierung	5 Minuten
Heartbeat	10 Minuten
Erdbebenalarm	Bei Auftritt eines Alarms

#### 7.4.3 Klassendiagramm Localizer

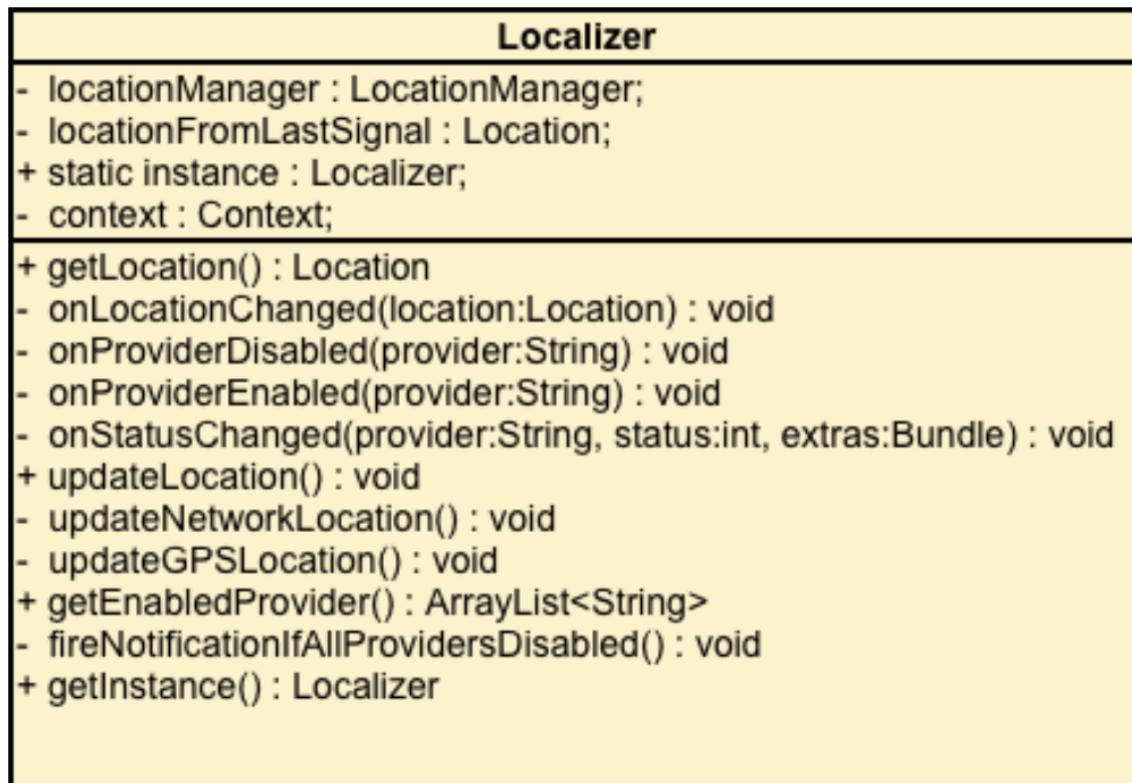


Abbildung 27: Lokalisierung: Localizer

## 8 Kommunikation (Benjamin Brandt)

Die Kommunikation der App mit dem Server läuft bei der Kommunikation der App mit dem RESTful Webservice über Http Befehle und für das Senden von Alarmnachrichten an die verschiedenen Geräte wird Google Cloud Messaging verwendet.

### Google Cloud Messaging

Der Service Google Cloud Messaging (GCM) wird Entwicklern von Android Apps kostenlos von Google zur Verfügung gestellt. Er erlaubt diesen über bestehende Google Server Infrastruktur Daten an ihre Apps und von diesen an ihren Server zu senden. Hierbei übernimmt der GCM Service alle Aufgaben die mit der Übermittlung der Nachricht zu tun haben.

Hier eine Auflistung der Hauptfunktionen von GCM:

- Senden von Nachrichten vom eigenen Server zu den Nutzergeräten
- Empfangen von Nachrichten der Nutzergeräte am Server
- Die App muss nicht laufen um Nachrichten zu empfangen
- Es gibt keine grafische Oberfläche sämtliche Aufgaben die mit der Nachricht zusammenhängen werden von der eigenen Anwendung gesteuert.
- Voraussetzung ist Android 2.2 Froyo mit installiertem Playstore

### Architektur

Die Architektur von Google Cloud Messaging teilt sich in drei Bereiche auf. Der erste sind die GCM Connection Server die die Nachrichten von den Servern des Entwicklers annehmen und diese an die Android App des Benutzers schicken. Der zweite Bereich ist der Server des Entwicklers der die Nachrichten für die Benutzer Apps an die Connection Server sendet. Als dritter Bereich natürlich die Benutzer-App die sich zum Empfang von GCM Nachrichten bei GCM registrieren muss um eine eindeutige ID zu erhalten und dadurch nur für sie bestimmte Nachrichten erhält.

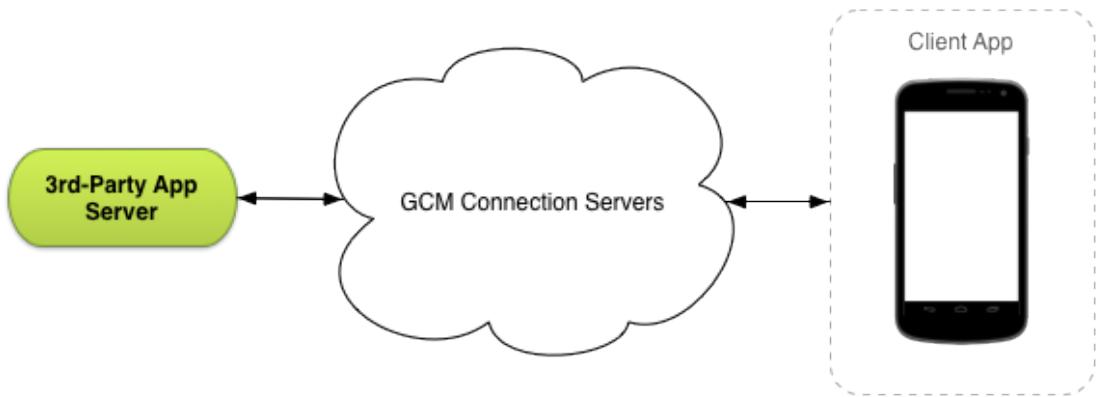


Abbildung 28: Struktur von GCM

## Android

Das Empfangen von Nachrichten in der App läuft über das Packet `GcmIntentService.java` hier werden die empfangenen Nachrichten nach ihrem Typ sortiert. Bei dem Messagetyp Send error wird der Fehler als solcher markiert ins Log geschrieben. Bei dem Messagetyp Deleted wird nach demselben Prinzip vorgegangen. Beim Typ Message wird die Prozedur `NotificationsService.sendAlarmNotification` aufgerufen. Hierdurch werden in Android die Notifications ausgelöst.

Listing 18: Empfangen von GCM Nachrichten unter Android

```

1
2   @Override
3   protected void onHandleIntent(Intent intent) {
4       Bundle extras = intent.getExtras();
5       GoogleCloudMessaging gcm = GoogleCloudMessaging.getInstance(this);
6       String messageType = gcm.getMessageType(intent);
7
8       if (!extras.isEmpty()) {
9
10           if (GoogleCloudMessaging.MESSAGE_TYPE_SEND_ERROR.equals(messageType))
11               {
12                   Log.i(TAG, "Send error: " + extras.toString());
13               }
14           else if (GoogleCloudMessaging.MESSAGE_TYPE_DELETED.equals(messageType)) {
15               Log.i(TAG, "Deleted messages on server: " + extras.toString());
16           }
17           else if (GoogleCloudMessaging.MESSAGE_TYPE_MESSAGE.equals(messageType)) {

```

```

17         NotificationsService.sendAlarmNotification(this,
18             extras.getString("message"));
19         Log.i(TAG, "Received: " + extras.toString());
20     }
21     GcmBroadcastReceiver.completeWakefulIntent(intent);
22 }
23 }
```

## Server

Der Server sendet ?? eine Benachrichtigung an die GCM Server diese leiten die Nachricht an die betroffenen Geräte-IDs weiter. Dies wird über einen REST-Methodenaufruf durchgeführt. Dieser wird an die Adresse <https://android.googleapis.com/gcm/send> gesendet, die Nachricht ist mit einem Authorisationskey ausgestattet damit nur Nachrichten vom richtigen Server gesendet werden können

Listing 19: Senden von Nachrichten vom Server

```

1
2 URL url = new URL(this.configuration.getApiUrl());
3     HttpURLConnection http = (HttpURLConnection) url.openConnection();
4     http.setDoOutput(true);
5     http.setRequestMethod("POST");
6     http.setRequestProperty("Content-Type", "application/json");
7     http.setRequestProperty("Authorization",
8         "key="+this.configuration.getAuthorizationKey()); //
9
10    OutputStreamWriter writer = new
11        OutputStreamWriter(http.getOutputStream());
12        writer.write(rawData.toString());
13        writer.flush();
```

Der Server empfängt 20, außerdem eine Bestätigung der Zustellung von den GCM Servern, diese enthält eine Zahl die kodiert ob die Nachricht angekommen und richtig interpretiert wurde. Bei Erfolg wird die Nachricht mit Code 200 bestätigt. Bei einer JSON Nachricht kann auch der Code 400 gesendet werden wenn z.B. ein anderer Datentyp als erwartet gesendet wurde. Wenn der Code 401 gesendet wird, ist die Authentifizierung am GCM Server nicht mit Erfolg abgeschlossen worden. Hiermit wird sichergestellt das Nachrichten nur vom richtigen Server gesendet werden können.

Listing 20: Empfangen von Bestätigungen am Server

```
1
2 BufferedReader reader = new BufferedReader(new
3     InputStreamReader(http.getInputStream()));
4
5     StringBuilder result = new StringBuilder();
6
7     for (String line; (line = reader.readLine()) != null; ) {
8         result.append(line);
9     }
10
11     writer.close();
12     reader.close();
```

## 9 Testing (Benjamin Brandt)

Das testen der Anwendung stellte uns zu Anfang vor einige Probleme die es zu lösen galt. Zum Beispiel: Wie können wir die Frequenz eines Erdbebens simulieren? Wie genau reagieren die Sensoren von verschiedenen Geräten auf das Erdbeben? Bestehen Anzeigeprobleme auf den verschiedenen Displaygrößen? Wie sieht die Darstellung auf dem Tablet aus? Wie stark beeinflusst die Performance der Geräte die Messung bzw. das Benutzererlebnis? Und nicht zuletzt wie hoch ist der Akkuverbrauch wenn die App im Hintergrund läuft.

### 9.1 Lösungsansätze

Zum lösen des Problems der Simulation eines Erdbebens fielen uns folgende Ansätze ein:

- Eine Rüttelplatte auf der die Geräte liegen
- Schütteln der Geräte bis sie auslösen
- Klopfen direkt auf das Gerät bzw. auf den Tisch neben dem Gerät

Zur Anpassung der Sensoren von verschiedenen Geräten könnte man die Frequenz der Messung so anpassen, dass im Mittel alle Geräte auslösen wenn sie zusammen auf einer Fläche liegen an der gerüttelt wird. Eine weitere Möglichkeit wäre die Geräte zu schütteln bis sie auslösen. Hierbei besteht aber das Problem, dass die Frequenz beim nacheinander Schütteln der Geräte nicht exakt die gleiche wäre.

Zum Messen ob die Performance des Gerätes sich auf die Darstellung in der App auswirkt gibt es Apps, die die aktuelle Framerate anzeigen oder als andere Möglichkeit, kann man sich auf sein subjektives Gefühl verlassen ob die App flüssig läuft.

Den Akkuverbrauch könnte man entweder mit einem extra hierfür programmierten Tool genau messen, oder mit den Bordmitteln von Android ungefähr bestimmen. Was aber nicht so genau wäre.

### 9.2 Durchführung

Zur Durchführung der Tests haben wir verschiedene Geräte eingesetzt. Hier eine Auflistung der Geräte mit ihren Spezifikationen.

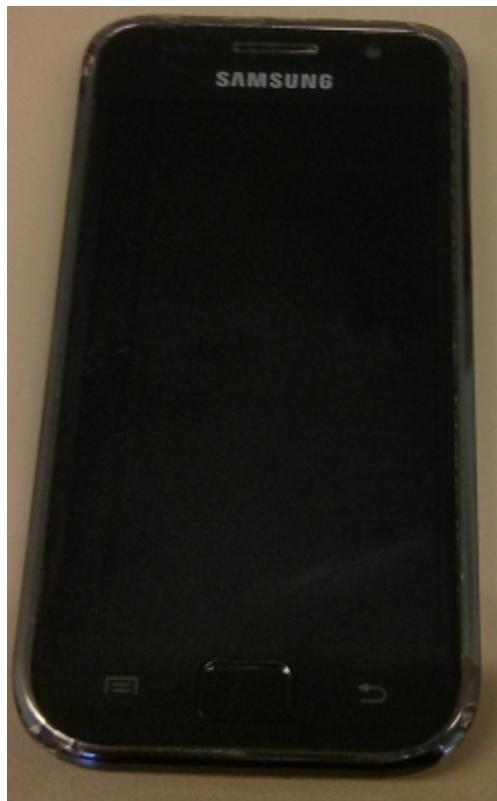


Abbildung 29: Galaxy S

Das Erste Gerät ist ein Samsung Galaxy S aus dem Jahr 2010 es besitzt einen 1Ghz Singlecore Prozessor und 512Mb RAM das Betriebssystem ist Android 4.4 als CustomROM.

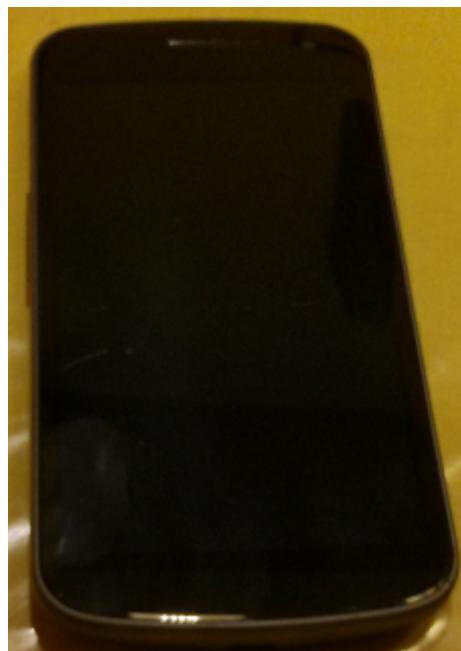


Abbildung 30: Galaxy Nexus

Das Zweite Gerät ist ein Google Galaxy Nexus aus dem Jahr 2011 es besitzt einen 1,2Ghz Dualcore Prozessor und 1Gb RAM das Betriebssystem ist Android 4.3.



Abbildung 31: Nexus 4

Das Dritte Gerät ist ein Google Nexus 4 aus dem Jahr 2012 es besitzt einen 1,5Ghz Quadcore Prozessor mit 2Gb RAM das Betriebssystem ist Android 4.4.



Abbildung 32: Acer Iconia A500

Das Vierte Gerät ist ein Tablet der Firma Acer aus dem Jahr 2011 mit einem 1Ghz Dualcore Prozessor und 1Gb RAM das Betriebssystem ist Android 4.1 als CustomROM.

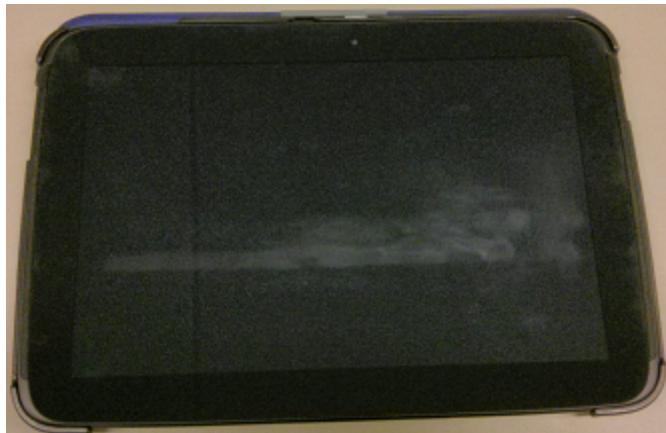


Abbildung 33: Nexus 10

Das fünfte Gerät ist ein Google Nexus 10 Tablet aus dem Jahr 2012 mit einem 1,7 Ghz Dualcore Prozessor und 2Gb RAM das Betriebssystem ist Android 4.4

Mit dieser Auswahl an Geräten lässt sich die aktuelle Verteilung der Android Geräte sowohl von der Leistung, sowie auch von der Performance gut abbilden. Bei den Tests stellte sich anfangs schnell heraus, dass die Aktualisierungsrate des Graphen in der Appansicht zu hoch war und dieser auf den schwächeren Geräten nicht mehr flüssig dargestellt werden konnte.

Überraschender Weise waren hingegen, die Sensordaten der verschiedenen verbauten Beschleunigungssensoren nicht so stark abweichend. Die größte Abweichung wurde durch ein Bumpercase verursacht das die Vibrationen zu stark abdämpfte. Ansonsten führte die eingestellte Frequenz von Anfang an zu recht passablen Erkennungsraten.

Des Weiteren musste die Darstellung auf den verschiedenen Displaygrößen getestet werden. Hierbei wurde festgestellt, dass auf Tablets der Raum absolut nicht ausgenutzt wurde. Hier wurde auf diese Feststellung hin mit der zusätzlichen Entwicklung einer extra Tablet Oberfläche reagiert.

Zum Testen der Alarmauslösung benutzen wir am Anfang die einfache Methode indem wir die Geräte solange schüttelten bis sie einen Alarm auslösten. Später haben wir sie auf einen Tisch gelegt und an diesem gerüttelt, als einfachen Ersatz für eine Rüttelplatte.

Im Späteren Verlauf des Projekts mussten wir außerdem das Auslösen der Alarme durch den Server testen. Hierzu sprachen wir uns über Whatsapp ab, wann wir die Geräte schütteln müssen und konnten somit auch die Prozentuale Schwelle die im Server implementiert ist testen.

Des Weiteren waren wir durch unsere Tests immer bemüht den Akkuverbrauch der App zu analysieren und zu optimieren. Hierzu beobachteten wir den Verlauf der Akkubenutzung anfangs mit GPS Ortung ständig aktiviert. Dieser war viel zu hoch und der Akku innerhalb weniger Stunden komplett entleert. Als nächste Option testeten wir mit einem GPS Aktualisierungintervall hier waren die Ergebnisse um einiges besser aber immer noch der Akkuverbrauch zu hoch. Als letzte Optimierung testeten wir einen Aktualisierungintervall der nur im Notfall auf GPS zurückgriff,

ansonsten aber die anderen Positionsbestimmungsmethoden wie Netz und WLAN nutzte. Hiermit war der Akkuverbrauch in passablen Bereichen und nur auf dem Land etwas erhöht da hier die Position über das Handynetz bestimmt wurde und somit teilweise über 3km Ungenauigkeit aufwies und deshalb GPS genutzt wurde.

Insgesamt sind die Tests gut verlaufen und brachten recht schnell gute Ergebnisse und bei Problemen beim Testen wurde immer schnell auf Verbesserungsvorschläge in der APP sowie im Webservice reagiert. Die Zusammenarbeit bei den Tests war sehr gut.

## **10 Ausblick**

Die aktuelle Implementierung ist noch in einem Alpha Status. Da uns weitreichendes Know-how im Bereich Erdbebenerkennung fehlte und ebenfalls technische Hilfsmittel, wie beispielsweise Rüttelplatten, nicht zur Verfügung standen, ist die Erkennung derzeitig nicht ausreichend präzise genug, um Erdbeben zuverlässig zu erkennen und Fehlalarme auszuschließen. Daher haben wir uns an die Bundesanstalt für Geowissenschaften und Rohstoffe gewandt, welche uns an das Geoforschungszentrum Potsdam verwies. Dort hatten wir Kontakt zu Herrn Stefano Parolai (Head of the Centre for Early Warning) hergestellt, der einige unserer Fragen beantwortete. Weiterhin arbeitet einer der Partner des GFZ Potsdam an einem Projekt, das in eine ähnliche Richtung führt wie dieses. Daher ist das GFZ Potsdam daran interessiert ein Skype Meeting abzuhalten. In diesem Meeting können weitere Fragen und eventuell gemeinsame Vorgehensweisen besprochen werden. Sollte es uns möglich sein die App zuverlässiger und präziser in der Erkennung zu implementieren, kann durchaus über eine Veröffentlichung nachgedacht werden.

## **11 Fazit**

Zusammengekommen ist in dem Projekt das gewünschte Ergebnis erreicht worden. Alle wesentlichen Anforderungen konnten umgesetzt werden. Das Projekt erforderte eine tiefgehende Auseinandersetzung in die Android und WebService Programmierung unter Java. Ebenso sind die Kompetenzen im Bereich der Teamarbeit bei allen Beteiligten erweitert worden.

Abschließend kann festgestellt werden, dass diese Lösung zur Erdbebenerkennung durchaus sinnvoll eingesetzt werden kann und eines Tages dazu dienen könnte, Sach- und Personenschäden bei einem Erdbeben zu minimieren.

# Literatur

- [1] ANDROID: *Android Developers - Location and Sensors API*. 2014. – URL <http://developer.android.com/guide/topics/sensors/index.html>. – Zugriffsdatum: 09.02.2014
- [2] ANDROID: *Android Developers - User Interface*. 2014. – URL <http://developer.android.com/guide/topics/ui/index.html>. – Zugriffsdatum: 09.02.2014
- [3] GOOGLE: *Collection of wifi data for Google location based services*. 2014. – URL [http://static.googleusercontent.com/media/www.google.com/de//googleblogs/pdfs/google\\_submission\\_dpas\\_wifi\\_collection.pdf](http://static.googleusercontent.com/media/www.google.com/de//googleblogs/pdfs/google_submission_dpas_wifi_collection.pdf). – Zugriffsdatum: 09.02.2014
- [4] GOOGLE: *Google Maps API v2*. 2014. – URL [https://developers.google.com/maps/documentation/android/start#getting\\_the\\_google\\_maps\\_android\\_api\\_v2](https://developers.google.com/maps/documentation/android/start#getting_the_google_maps_android_api_v2). – Zugriffsdatum: 09.02.2014

# Abbildungsverzeichnis

1	Smartphone Betriebssystem-Verbreitung . . . . .	6
2	Struktur des Projektes . . . . .	7
3	User Interface: Info . . . . .	10
4	User Interface: SHA-1 Fingerprint . . . . .	12
5	Google API Console nach erfolgreicher API Key Generierung . . . . .	12
6	Google Maps API Service aktivieren . . . . .	12
7	User Interface: Device Map . . . . .	13
8	User Interface: Map Types . . . . .	13
9	User Interface: Android Maps Extensions . . . . .	15
10	User Interface: Klassendiagramm Device Map . . . . .	17
11	User Interface: Klassendiagramm Device Map . . . . .	17
12	User Interface: Settings . . . . .	18
13	Beschleunigungssignal nach der Zusammenfassung . . . . .	22
14	Beschleunigungssignal mit Schwellwerten . . . . .	22
15	Beschleunigungssignal mit Nulldurchläufen analysiert . . . . .	23
16	Lokalisierung: Lokalisierung mit NETWORK Provider . . . . .	27
17	Lokalisierung: User Interface bei Nutzung des NETWORK Providers . . . . .	28
18	Lokalisierung: Timing-Advance Verfahren . . . . .	29
19	Lokalisierung: NETWORK Provider in der Praxis . . . . .	29
20	Lokalisierung: GPS 2 Satelliten . . . . .	30
21	Lokalisierung: GPS 3 Satelliten . . . . .	30

22	Lokalisierung: App Permissions . . . . .	31
23	Lokalisierung: Update Interval . . . . .	34
24	Lokalisierung: Test auf der Autobahn . . . . .	35
25	Lokalisierung: Notification in der App . . . . .	37
26	Lokalisierung: Notification bei geschlossener App . . . . .	37
27	Lokalisierung: Localizer . . . . .	38
28	Struktur von GCM . . . . .	40
29	Galaxy S . . . . .	44
30	Galaxy Nexus . . . . .	44
31	Nexus 4 . . . . .	45
32	Acer Iconia A500 . . . . .	45
33	Nexus 10 . . . . .	46