UNIKASSEL VERSITÄT
Fachbereich Elektrotechnik/Informatik
Fachgebiet Softwareengineering
Research Group Software Engineering

Lokale Wlan baiserte Mehrspieler Bibliothek für Android

BACHELOREARBEIT

zur Erlangung des Grades eines Bachelore-Informatikers
im Fachbereich Elektrotechnik/Informatik
der Universität Kassel

eingereicht am	13.06.2013
betreut von	Stefan Lindel
bei	Prof. Dr.-Ing. Albert Zündorf
	Prof. Dr.-Ing. Lutz Wegner
	Universität Kassel

von	Alexander Gerb
	Liegnitzerstr. 6
	34123 Kassel

Zusammenfassung

Diese Arbeit umfasst die Implementierung einer Bibliothek für Multiplayerspiele auf Peer-to-Peer-Basis [3]. Peer-to-Peer bedeutet das für die Kommunikation kein Server verwendet wird. Dies wird jedoch noch genauer im Kapitel 2.2 erläutert. Für die Verbindung zwischen den Geräten wurde das Rahmenwerk AllJoyn [2] verwendet, welcher im Kapitel 2.4 erläutert wird. Weiterhin wurden zwei Spiele mithilfe dieser Bibliothek implementiert und dienen als praktische Beispiele. Die Bibliothek ist für Android Geräte bestimmt und wird daher auf Basis des Android SDK [1] realisiert. Die Bibliothek sollte eine Kommunikation zwischen mehreren Geräten soweit vereinfachen, dass bei der Implementierung weiterer Spiele nur um die Spielmechanik Gedanken gemacht werden muss. Selbst die Lobby-Funktionalität, wie das Erstellen und das Verbinden zum Spiel, wird von der Bibliothek übernommen. Im weiteren Verlauf wird die implementierte Bibliothek als ***PTPLibrary*** genannt.

Schlagwörter: Multiplayer, Android, Lokal, Peer-to-Peer

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig angefertigt und bis auf die beratende Unterstützung meines Betreuers, fremder Hilfe nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß veröffentlichtem oder unveröffentlichtem Schrifttum entnommen sind, habe ich als solche kenntlich gemacht.

Kassel, xx.xx.20xx

Alexander Gerb

Inhaltsverzeichnis

1 Einleitung

Spiele auf mobilen Endgeräten gibt es unzählige. Jedoch sind die meisten dieser Spiele entweder Solospiele, also Spiele für einen Spieler, oder sind für mehrere Spieler, aber unter Verwendung eines Servers. Trotz der mittlerweile gut ausgebauten Netzabdeckung und guter Internetgeschwindigkeit auf Smartphones, kann das Spielerlebniss durch eine schwankende Internetgeschwindigkeit getrübt werden. So sind z.B. Actionspiele meist nur über eine Wlan Verbindung spielbar. Außerdem gibt es Situationen bei denen man keine oder nur eine beschränkte Internetverbindung hat, z.B. wenn man mit dem Zug unterwegs ist oder sich gerade in Ausland befindet und sich die Roaminggebühren sparen will. Um die Entwicklung solcher Spiele vorran zu treiben und es den Entwickler soweit wie möglich zu erleichtern, sollte im Rahmen dieser Arbeit eine Bibliothek entwickelt werden. Es sollen die kommunikatonsbedingten Herausforderungen mit der PTP-Library gelöst werden, sodass die Entwickler sich nur um die Implementierung der Spiele selbst gedanken machen müssen. Die Entwickler hätten nur die Aufgabe sich um die Synchronisierung der spielbezogenen Daten zu kümmern. Die Übertragen der Daten an die einzelnen Gerät würde von der PTP-Library übernommen. Damit die Entwicklung solcher Spiele für die Hobbyentwickler keine finanziellen Hürden stellt, wurde die PTP-Library mit dem Android SDK [1] entwickelt. Da das Android SDK kostenlos für jeden Entwickler zur Verfügung steht, kann jeder Entwickler, der über ein Android Gerät verfügt gleich mit der Entwicklung des Spieles loslegen. Natürlich lässt sich die PTP-Library auch für praktische Anwendungen, wie z.B. Textmessanger, verwenden, jedoch wird im Rahmen dieser Ausarbeitung nur auf die Realisierung von Spielen eingegangen.

1.1 Lösungsansatz

Um sich die ganzen Herausforderung bei der Realisierung von Peer-to-Peer [3] Kommunikation zu ersparen wurde ein weiteres Rahmenwerk namens AllJoyn verwendet, welches die die Verbindung und die Kommunikation zum größten Teil schon übernimmt. Somit blieb zum Einen eine geschickte Integration von AllJoyn in die

PTP-Library. Weiterhin mussten die vielen, gerade auf den ersten Blick komplizierten, Konfigurationen des AllJoyn Rahmenwerks durch eine einfache Schnittstelle erweitert werden. Die Netzwerkverbindung läuft in einem seperaten Thread und muss daher mithilfe von ***Handlern*** 2.3.6 von dem ***UI-Thread*** 2.3.7 abgekoppelt werden, sodass sich die beiden Threads nicht behindern. Als praktische Anwendungsbeispiele wurde ein Echtzeitpuzzlespiel und ein Zugbasiertes Kartenspiel namens MauMau implementiert.

1.2 Gliederung

Die Arbeit beinhaltet zu Anfang die Erklärung der verwendeten Rahmenwerke AndroidSDK und AllJoyn. Dabei werden die Konzepte dieser Rahmenwerke grob erläutert. Dann wird das Konzept und die Implementierung der PTP-Library, welches das Hauptaugenmerk dieser Arbeit ist, beschrieben. Daraufhin wird beschrieben, wie die zwei Spielebeispiele realisiert wurden, sowie Herausforderung, die während der Implementierung entstanden.

2 Grundlagen

2.1 Konzepte

Die PTPLibrary basiert auf dem Peer-to-Peer Prinzip und benötigt deswegen keinen Server. Jedes Gerät ist gleichgestellt und interagiert über einen ***Bussystem*** mit den anderen Geräten. Über eine Schnittstelle kann der Entwickler Nachrichten an alle anderen Teilnehmer versenden und eingehende Nachrichten empfangen. Jedes der Gerät hat eine identische Version des Spiels, sodass die Organisation bei der Verbindung realisiert wird. Wie in der Abbildung 2.1 zu sehen, kann jedes Spiel auf einem Gerät mit den anderen Gerät eine Verbindung über einen ***AllJoyn Bussystem*** 2.4.1 aufbauen. Das Spiel selbst interagiert nur mit der ***PTPLibrary***, welche selbst eine Logische Verbindung aufbaut und die Nachrichten entsprechend weiterleitet. Ein Teilnehmer kann eine Session erstellen, zu der sich die die anderen Teilnehmer verbinden können um an der Spielrunde teilnehmen zu können. Durch die Session wird die Möglichkeit gegeben mehrere Spiele simultan im selben Netzwerk zu spielen.

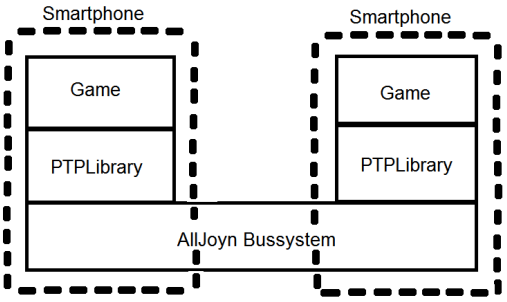
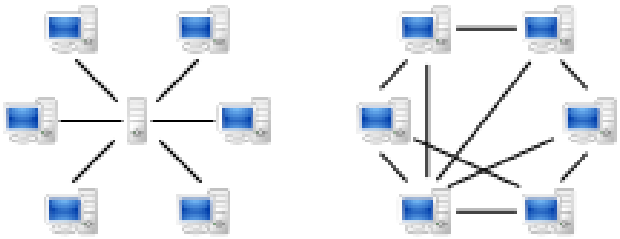


Abbildung 2.1: Architektur-Konzept

2.2 Peer-to-Peer

Peer-to-Peer kommt vom englischen Wort peer=Gleichberechtigter und entspricht somit einem Netzwerk aus gleichberechtigten Teilnehmern. Da es bei dem Peer-to-Peer Netzwerk im Grunde keinen Server gibt, welcher sonst die Steuerung der Prozesse und der Ressourcen übernimmt, müssen die einzelnen Peers sich selbst darum kümmern und sich entsprechend selbst organisieren.



(a) Server (b) Peer-to-Peer

Abbildung 2.2: (a) Server basiertes vs. (b) Peer-to-Peer basiertes Netzwerk

Die Bilder 2.2 ***a*** und ***b*** verdeutlichen nochmal das Gesagte. Durch das Weglassen eines fest zugeordneten Servers, erhält man eine Flexibilität, sodass nur Peers benötigt werden um das Spiel zu spielen. Es ergeben sich jedoch auch Probleme bei dieser Art der Realisierung, die man mit einem serverbasiertem system kaum hätte. Normalerweise bei einem serverbasiertem Spiel wird der Zustand des Spiels auf dem Server geregelt und alle zu dem Server verbundenen Geräte brauchen den Server nur nach diesen Zustand zu fragen. Außerdem können rechenintensive Aufgaben vom Server übernommen werden und so die einzelnen Geräte entlasten. Bei einem Peer-to-Peer System hingegen muss der Zustand von jedem einzelnen Gerät ermittelt werden und mit den anderen Geräten synchonisiert werden. Dies hat einen höheren Ressourcenaufwand und limitiert somit die Anzahl der möglichen Teilnehmer, da die Informationen mit allen Peers ausgetauscht werden müssen. Jedoch ist es auch sehr vom Anwendungfall abhängig, denn ein entsprechendes Gegenbeispiel für eine hohe Peersanzahl zeigten Tauschbörsen wie Torrent oder Napster um nur zwei bekannte Beispiele zu nennen. Somit liegt die Herausforderung bei den Entwicklern, da sie zum einen entscheiden müssen wie viele Spieler maximal zulässig sind und somit auch die Anforderung für die Komplexität des Spiel setzen.

2.3 Android SDK

Android ist das Linux-basierte Betriebssystem für mobile Endgeräte, welches von Google 2011 offiziell zur Verfügung gestellt wurde. Android selbst gilt als sogenannte freie Software, welche bis auf den System-Kern unter der Apache-Lizens steht. Diese Tatsache unter Anderen ermöglichte eine rasante Verbreitung dieses Betriebsystems auf vielen Geräten unterschiedlicher Hersteller. Somit waren im Mai 2013 etwa 900 Millionen Android End-Geräte aktiviert [8] und man merkt schnell, dass

die Popularität dieses Betriebssystem immer mehr zunimmt. Google hat eine Entwicklungswerkzeugsammlung zur Verfügung gestellt, welche die Entwicklung von Applikationen für Android möglichst einfach gestallten soll. Bei dieser Werkzeugensammlung handelt es sich um das Android SDK, welches auch als das Android Developer Tool kurz ADT verfügbar ist. ADT ist ein Plugin für das mittlerweile sehr weit verbreitete Entwicklungsumgebung Eclipse, welche die Entwicklung und die Übertragung der Applikation auf das Gerät mühelos ermöglicht. Weiterhin bringt das Android SDK einen Emulator mit sich, welches das Testen von Apps unter unteschiedlichen Konfigurationen ermöglicht ohne dass man ein Android-Gerät benötigt. Vorallem ermöglicht das Android SDK die Entwicklung der Apps in der Programmiersprache Java, welche sich immer höherer Beliebtheit erfreut. Als nächstes wird auf die einzelne Grundlagen von Android SDK eingegangen um die Funktionalität dieser zu beschreiben.

2.3.1 Activity

Eine Activity ist eine Klasse, welche die Erstellung von einzelnen UI-Fenstern übernimmt. Somit bestehen die meisten Apps in Android aus mehreren Activities, welche mit einander verbunden sind. Eine selbst erstellte Activity muss von der Klasse ***Activity*** erben. Zusätzlich muss die Methode ***onCreate()*** in dieser Klasse überschrieben werden. Diese Methode wird jedesmal aufgerufen wenn die Activity erstellt wird und auf dem Bildschirm angezeigt werden soll. In diese Methode könne Aufrufe von Fenstern kommen, die die UI beinhalten, oder andere Operationen die beim Start notwendig sind.

2.3.2 Service

Ein Service ist eine Komponente, welche dazu gedacht ist Hintergrundprozesse zu übernehmen. Ein Service wird von z.B. einer Activity gestarted und läuft dabei im Hintergrund, selbst wenn die Activity nicht mehr existiert. Somit bietet sich ein Service gut an um z.B. die Netzwerkkommunikation im Hintergrund zu behandeln ohne die Applikation selbst zu behindern. Weiterhin lässt sich ein Service auch an z.B eine Applikation binden, sodass der Service auch beendet wird wenn die Applikation geschlossen wird.

2.3.3 Application

Application bietet zusätzlich zu den Activities die Möglichkeit während der ganzen Laufzeit der Applikation eine feste Instanz zu haben, welche den Zustand bestimmter Daten beinhaltet. So kann man es im Prinzip mit einem Singleton vergleichen der den Status der Applikation hält. Um an die Instanz zu kommen muss man aus dem Kontext heraus die Methode ***Context.getApplicationContext()*** aufrufen.

2.3.4 Context

Der Context beinhaltet Informationen über die Applikationsumgebung und lässt verschiedene Aktionen zu, wie z.B. das Aufrufen von weiteren Activities. Eine Activity ist eine Unterklasse vom Context und wird bei der Erstellung von z.B. einer View an diese übergeben.

2.3.5 View

Eine View representiert eine Sammlung von UI-Elementen auf einem Bildschirm. Die Elemente werden meist über das XML-Layout erstellt und darüber referenziert. Activities haben so die Möglichkeit eine Unterklasse der View als Instanz aufzurufen, welche das UI-Fenster representiert oder können die UI-Elemente direkt über das XML-Layout laden.

2.3.6 Handler

Ein ***Handler*** ist ein Object, welche bei der Instantiierung an einen Thread gebunden wird. Werden Nachrichten an diesen Handler geschickt, so werden diese Nachrichten von dem an diesen Handler gekoppelten Thread ausgeführt. Der Handler ist somit gut für die Kommunikation zwischen verschiedenen Thread geeignet.

2.3.7 UI-Thread

Der UI Thread ist der Hauptthread, welcher beim Start der Applikation gestartet wird. Dieser Thread ist für die Darstellung von UI-Elemente verantwortlich sowie auf Benutzeraktivitäten wie Touchevents zu reagieren und sie zu verarbeiten.

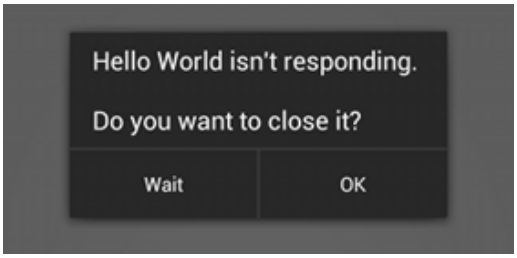


Abbildung 2.3: ANR

Es ist wichtig langwierige und blockende Aufgaben wie die Netzwerkkommunikation in ein seperaten Thread auszulagern um den UI Thread nicht zu überlasten. Ein blockierter UI Thread kann schnell zu einer sogenannten ANR-Meldung führen, wie im Bild 2.3 zu sehen. Außerdem muss bei Verwendung von mehreren Thread darauf geachtet werden, dass die UI Komponenten nur vom UI Thread angefasst werden dürfen. Es gibt die Möglichkeit bei Background Thread die ***runOnUiThread()*** Methode zu verwenden, welche die auszuführende Aufgabe an den UI Thread übergibt.

2.3.8 AndroidManifest

Das Android Manifest ist eine XML-Datei, welche sich im Wurzelverzeichniss des Projektes befindet. Sie beschreibt welche Rechte die Anwendung benötigt, sowie alle Activities und Services, die während der Laufzeit gestartet werden.

```
1  <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" >
2  </uses-permission>
```

Codeausschnitt 2.1: Android Manifest XML

Im Ausschnitt 2.1 wird das Recht, den Zustand der WiFi Verbindung zu erfragen, erteilt. Weiterhin beinhaltet das Manifest, die Information über die verwendete Android SDK Version, das Icon der Application, sowie Themes und andere relevante Daten.

2.4 AllJoyn

AllJoyn ist ein Open-Source Peer-to-Peer Rahmenwerk welches es erlaubt Verbindungen zwischen verschiedenen Geräten herzustellen. AllJoyn wird von Qualcomm Innovation Center Inc. entwickelt und steht unter der Apache v.2 Licenz. Besonderer Augenmerk dieses Framework besteht in der Tatsache, dass es verschiedene Betriebssysteme unterstützt und eine Vielzahl von Programmiersprachen, darunter C♯, C++, Java sowie Objective C. Somit lassen sich Anwendungen auf Peer-to-Peer Basis auf Windows,MacOS,Linux,Android und iOS entwickeln und miteinander verbinden. Es bietet weiterhin die Unterstützung für Bluetooth, Wifi und Ethernet. Vorallem ermöglicht AllJoyn die Verbindung von Mobilen-Endgeräten im Wifi-Netz ohne sich selbst um das Finden und Verbinden kümmern zu müssen. Das AllJoyn Rahmenwerk übernimmt somit alle Aufgaben auf der Netzwerkschicht. Außerdem bietet AllJoyn für jedes Betriebssystem ein SDK an, welches alle notwendigen Bibliotheken sowie einige Beispielanwendungen beinhaltet.

2.4.1 Bussystem

Das Prinzip von AllJoyn basiert auf einem Bussystem zu dem sich einzelne Anwendung verbinden können und ist in der Abbildung 2.4 nochmal visuell verdeutlichts. Jedes dieser Anwendung muss einen ***BusAttachement*** erstellen und den entsprechenden EventHandler implementieren über den dann die Nachrichten von anderen Anwendungen behandelt werden. Es können Methoden direct auf einem entfernten Object ausgeführt werden. Diese Objekte werden in der Abbildung als ***ProxyObject*** dargestellt. Es könenn auch Nachrichten an alle Teilnehmer geschickt werden, welche von einem ***EventHandler*** behandelt werden. Der ***EventHandler*** muss das entsprechende Interface implementieren, welcher die Mehoden definiert, über die kommuniziert wird. Das genauere Prinzip des ***BusInterface*** wird weiteren Verlauf genauer erklärt. Der Bus selbst ist ein sogenannter Daemon, also ein Hintergrundservice, welcher auf dem Gerät läuft und sich um netzwerktechnischen Aufgaben kümmert. Dieser wird z.B. über einen Aufruf in der

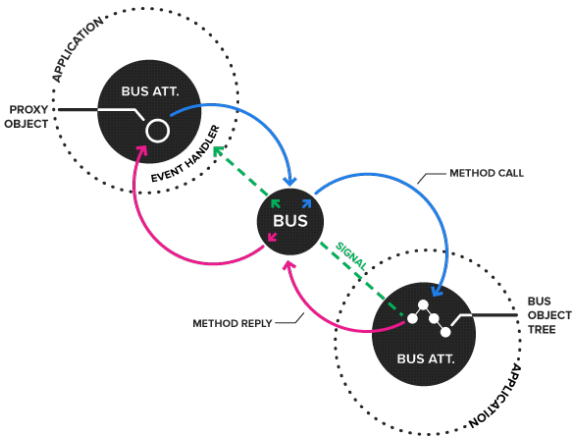


Abbildung 2.4: Bussystem [9]

Anwendung gestartet und bietet durch das BusAttachement eine Schnittstelle für Anwendungen um damit zu interagieren.

2.4.2 BusAttachement

Um mit dem Bussystem zu kommunizieren, muss man einen ***BusAttachement*** erstellen, wie im Codeausschnitt 2.2 dargestellt.

```
1 bus = new BusAttachment("name",BusAttachment.RemoteMessage.Receive)
```

Codeausschnitt 2.2: BusAttachement

Der ***name*** Parameter ist notwendig um verschiedene BusAttachements zu den entsprechenden Anwendungen zuzuordnen, falls mehrere Anwendungen über den selben Bus kommunizieren. Es ist nämlich möglich mehrere Anwendungen mit AllJoyn auf dem selben Gerät laufen zu lassen und dass sie miteinander kommunizieren. Als Zweitparameter lässt sich noch angeben ob das BusAttachement eingehende Nachrichten verwerfen oder behandeln soll. Durch den Parameter ***BusAttachement.RemoteMessage.Receive*** wird aufgefordert, eingehende Nachrichten nicht zu verwerfen, sondern an die jeweiligen SignalHandler weiterzuleiten.

Die Abbildung 2.5 verdeutlicht nochmal die Funktionsweise des BusAttachements. Die mit den Buchstaben ***A,B,C*** dargestellten Rechtecke representieren jeweils einen BusAttachements, welche zu einen gemeinsame Bus verbunden sind, welcher durch eine dicke durchgezogene Linie dargestellt ist. Dieser Bus stellt intern eine logische Verbindung mit den anderen Smartphone her, sodass jedes BusAttachement mit anderen BusAttachements kommunizieren kann.

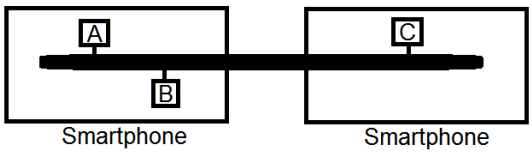


Abbildung 2.5: Bus

2.4.3 Interface

Die Kommunikation bei AllJoyn funktioniert über Interfaces, also eine konkrete Beschreibung der Methoden über die Nachrichten verschickt werden.

```
1 @BusInterface (name = "de.package.MyInterface")
2 public interface MyInterface{
3 @BusMethod
4 public void MyMethod() throws BusException;
5 @BusSignal
6 public void MySignal() throws BusException;
7 }
```

Codeausschnitt 2.3: BusInteface Annotation

Im Codeausschnitt 2.3 ist ein ***BusInteface*** dargestellt. Dieses Inteface muss eine Reihe an Annotation beinhalten. Die erste Annotation in Zeile 1 beschreibt das Inteface, indem es den vollständigen Namen nochmal aufführt. Die Methode ***MyMethod()*** ist eine Methode, welche auf einem ***ProxyObject*** von einem anderen Gerät aus ausgeführt werden kann. Die Methode ***MySignal()*** hingegen definiert die Methode für den ***SignalHandler***, welcher auf die Nachrichten reagiert, die an alle Teilnehmer gleichzeitig geschickt werden.

2.4.4 ProxyObjecte

Bei Proxy Objekten handelt es sich um Objekte die dazu benutzt werden um Methoden auf Objekten anderer Geräte oder Anwendungen aufzurufen. Sie implementieren das zuvor definierte Interface, worüber dann die Kommunikation realisiert wird. Es lässt sich z.B. das ProxyObject vom Inteface ***MyInteface*** aus dem Codeausschnitt 2.3 vom BusAttachement bekommen und darauf die Methoden, welche im Inteface definiert sind ausführen.

2.4.5 BusObject

Ein BusObject ist ein Objekt, welches ein definierte Interface implementiert und unter einem bestimmten Pfad abgespeichert wird. Der Pfad ist eine busweiter einzigartige Zeichenkette, die in Form eines Dateipfades dargestellt wird. Über den diesen Pfad kann über das BusAttachement das entsprechende ProxyObject erfragt werden um darauf Methoden auszuführen.

2.4.6 SignalHandler

Ein SignalHandler ist ein Object welches die Behandlung von eingehenden Nachrichten implementiert. Der Unterschied von einem BusObject und einem SignalHandler ist, dass beim BusObject man nur die Methode auf dem entsprechenden Object ausführt, welcher unter einem bestimmten Pfad gespeichert ist. Es wird somit immer auf einem Object die Methode ausgeführt. Ein SignalHandler reagiert auf Methodenaufrufe, die an alle Teilnehmer geschickt werden. Dies lässt sich mit einem Broadcast vergleichen. Falls es sich bei dem Inteface um einen SignalHandler handelt müssen die Methoden @BusSignal als Annotationen beinhalten Zusätzlich muss man bei der Implementierung eine weitere Annotation hinzufügen um den Nachrichtentyp explizit zu definieren.

```
1 @Override
2 @BusSignalHandler(iface = "de.packege.MyInterface", signal = "MyHandler")
3 public void MyHandler() throws BusException {}
```

Codeausschnitt 2.4: BusSignalHandler Annotation

Im Codeausschnitt 2.4 ist die notwendige Annotation aufgeführt. Diese beinhaltet den vollständigen Intefacenamen und den Namen der Methode.

2.4.7 SignalEmitter

Ein SignalEmitter ist das Objekt das dazu verwendet wird Nachrichten an alle Teilnehmer zu senden. Es ist im Prinzip das ProxyObject zu allen Teilnehmern.

```
1 emitter = new SignalEmitter(busObject, id, SignalEmitter.GlobalBroadcast.Off);
2 myInterface = (MyInterface) emitter.getInterface(MyInterface.class);
```

Codeausschnitt 2.5: SignalEmitter

Der Codeausschnitt 2.5 veranschaulicht, wie ein SignalEmitter erstellt wird. Bei der id handelt es sich um die SessionID, welche beim Verbinden zu einer Session mitgeteilt wird. Zusätzlich lässt sich noch konfigurieren ob das Signal auch über den Bus hinaus weitergeleitet wird, falls jemand zu mehreren Bussystemen verbunden ist. Als Standartkonfiguration ist die Signalweiterleitung aus. In Zeile 2 wird mit Angabe des Interfaces ein ProxyObject erstellt.

2.4.8 Session

Eine Session ermöglicht es mehrere Teilnehmer zu einer gemeinsamen Einheit zusammenzufassen. Dies erlaubt es verschiedene Anwendungsabläufe wie z.B. eine Spielrunde von einander getrennt zu handhaben. Dazu muss ein ***Channel*** erstellt werden, wohin sich alle anderen Teilnehmen verbinden können. Dazu benötigen alle Geräte zum einen den Namen des Channels und zum anderen die ***Channelportnummer***. Dazu kann der Host den erstellten Channelname den anderen Teilnehmern, über das ***Advertising*** mitteilen. Um diese Mitteilungen auch zu empfangen, müssen die Teilnehmer einen ***BusListener*** implementieren, welcher dann auf solche Nachrichten horcht. Das Erstellen der Session benötigt unter einigen Einstellungen, wie die Portnummer,Transportprotokol usw, auch den SessionPortListener, welcher z.B. das Verbinden von anderen Teilnehmern behandelt und bestimmte Aktionen dann ausführen kann. Typische Methoden eines SessionPortListeners sind im Codeausschnitt 2.6 aufgeführt.

```
1 public boolean acceptSessionJoiner(short port,String joiner, SessionOpts opts)
2 public void sessionJoined(short sessionPort, int id, String joiner)
```

Codeausschnitt 2.6: SessionPortListener Methoden

Durch die Implementierung der aufgeführten Methoden, lassen sich die Verbindungen zu der Session kontrollieren.

3 Implementierung

Es gibt viele verschiedene Spielprinzipien und daher sollte soweit es geht mit der PTPLibrary den späteren Entwicklern so viel Freiheit gelassen werden wie möglich. Jedoch gibt es viele Abläufe, die sich immer wieder wiederholen. So hat man den typschen Ablauf, dass man das Spiel startet und man sich in einer Lobby befindet. Danach kann man entweder ein Spiel erstellen und sich zu einem bereits erstellten Spiel verbinden. Diesen Ablauf wird durch die PTPLibrary übernommen, sodass man sich hinterher darum nicht mehr kümmern muss. Der Entwickler hätte dann nur noch die Aufgabe sich um die Activities zu kümmern, die das Spiel representieren und die Verbindung und Vermittlung würde die PTPLibrary übernehmen. Um das Bibliothek benutzen zu können müsste der Entwickler dann die PTPLibrary als Bibliothek in sein Projekt einbinden und er könnte loslegen.

3.1 PTPHelper

Der PTPHelper ist eine Schnittstelle zwischen dem Spiel und dem Hintergrundservice, welcher sich um die Verbindung kümmert. Nach Abwägen mehrerer Möglichkeiten wie man den Service, welcher sich um die Verbindungstechnischen Abläufe kümmert, in die Applikation integrieren kann, ist die Wahl schließlich auf die Benutzung eines Singletons gefallen. Zwar ließe sich eine abstrakte Application benutzen, die der Entwickler in seinem Projekt implementieren müsste, aber dies würde dem Entwickler auch eine Architekturentscheidung aufzwingen. Außerdem ist eine zu enge Bindung an das Android Rahmenwerk unvorteilhaft, da sich dessen API recht oft ändert und die Entwicklung in der Zukunft erschweren kann. Durch das Singleton kann der Entwickler entscheiden, wann er den Service startet und wie er damit interagieren will. So kann er z.B. den Einzelspielermodus vollkommen ohne der Benutzung des PTPHelpers realisieren und erst bei Multiplayermodus den Service starten. Um den PTPHelper zu initialisieren muss der Entwickler in seinem Code den folgenden Befehl aus dem Codeausschnitt 3.1 ausführen.

```
1 PTPHelper.initHelper(MyInterface.class, context, proxyObject, signalHandler,
      MyLobby.class);
```

Codeausschnitt 3.1: Helper initialisieren

Daraufhin kann er über den Getter an die initialisierte Instanz kommen, wie im Codeausschnitt 3.2 gezeigt.

```
1 PTPHelper.getInstance();
```

Codeausschnitt 3.2: Getter für die initialisierte Instanz

Man muss jedoch beachten, das der Hintergrundservice noch nicht vollständig gestartet werden kann, da es sich hierbei um einen asynchronen Ablauf handelt. Um den Zustand des Hintergrundservices zu erfahren kann man über den Methodenaufruf, wie im Codeausschnitt 3.3 gezeigt, erfragen.

```
1 PTPHelper.getIntance().getConnectionState();
```

Codeausschnitt 3.3: Verbindungstatus erfragen

Dabei handelt es sich um einen Integer, welcher die Werte CONNECTED=7 und DISCONNECTED=8 annehmen kann. Bei der Initialisierung wird der P2PService im Hintergrund gestartet welcher die Verbindung zum Bussystem aufbaut, daher sind die ganzen Parameter notwendig. Es wird zum einen die Interface Klasse benötigt über das die Kommunikation mit den anderen Peers realisiert wird und gibt den Typ für die generische P2PHelper Klasse vor. Der context ist die Activity oder die Application an die der Service gebunden wird. Der MyLobby.class Paramater übergibt die Klasse, die die AbstractLobbyActivity aus dem PTPRahmenwerk implementiert. Die LobbyActivity ist an den Service gebunden, da sie über den Zustand der Verbindung benachrichtigt wird. Die genauere Funktionalität der Lobby wird später noch erklärt. Das ProxyObject und der SignalHandler müssen gleich bei der Initialisierung übergeben werden, da sie bei der Verbindung mit dem Bus beim Bus registriert werden müssen, damit sie gleich auf Nachrichten reagieren können und das ProxyObject gleich einem konkreten Endpunkt zugewiesen werden kann. Damit der Service nur so lange läuft wie die Application wird der Service gebunden an diese gebunden, siehe Codeausschnitt 3.4.

```
1 Intent service = new Intent(context,PTPService.class);
2 boolean bound = context.bindService(service,new PTPServiceConnection(),Service.
      BIND_AUTO_CREATE);
```

Codeausschnitt 3.4: Service binden

Bei dem PTPServiceConnection Objekt handelt es sich im Grunde nur um einen Listener, welcher benachrichtigt wird, wenn die Verbindung zum Service aufgebaut oder getrennt ist. Der Flag ***Service.BIND_AUTO_CREATE*** besagt, dass nach dem Binden des Services der Service automatisch über die Methode ***onCreate()*** erstellt werden soll.

Die PTPLibrary ist dazu ausgelegt nur im WLAN zu funkionieren, somit wird bei der Initialisierung geprüft ob das Gerät über eine WiFi Verbindung verfügt. Dies wird über den WiFiManager realisiert, wie im Codeausschnitt 3.5 dargestellt.

```
1 WifiManager wifiManager =(WifiManager)context.getSystemService(Context.
      WIFI_SERVICE);
2 WifiInfo currentWifi = wifiManager.getConnectionInfo();
3 if((currentWifi==null || currentWifi.getSSID()== null || currentWifi.getSSID().
      isEmpty()){
4   //Zeige, dass es keine WiFi Verbindung gibt
5 }
```

Codeausschnitt 3.5: WifiManager

Im Falle dessen, dass es keine WiFi-Verbindung gibt, wird die Initialisierung abgebrochen und der Service wird nicht gestartet. Android ermöglicht es zusätzlich ein AccessPoint zu erstellen, worüber sich andere Geräte verbinden können. Die Information über diesen Zustand wird jedoch nicht ohne Weiteres über den WiFi-Mananger herausgegeben, sodass man an diese Information nur über Java-Reflection gelangt. Wie das funktioniert wird im Codebeispiel 3.6 verdeutlicht.

```
1 Method method = wifiManager.getClass().getMethod("isWifiApEnabled");
2 state = (Boolean) method.invoke(wifiManager);
```

Codeausschnitt 3.6: Access Point Status erfragen

Nach dem jedoch der Helper initialisiert ist und der Service gestartet, kann man über den Helper den Service mitteilen sich z.B. zu einem Channel zu verbinden oder einen Channel zu erstellen. Dies geschickt über einfache Methodenaufrufe, wie im Beispiel 3.7 gezeigt.

```
1 //Host
2 PTPHelper.getInstance().setHostChannelName(name)
3 PTPHelper.getInstance().hostStartChannel()
4 ...
5 //Client
6 PTPHelper.getInstance().setClientChannelName("channelName");
7 PTPHelper.getInstance().joinChannel();
```

Codeausschnitt 3.7: Host/Client Sessions Handhabung

Der PTPHelper erlaubt es zusätzlich verschiedene Observer zu registrieren um auf möglichliche Events entsprechend reagieren zu können. Zu den Observern gehören die Interfaces LobbyObserver, BusObserver, SessionObserver und der HelperObserver, welche der Entwickler selber Implementieren kann. Zu den LobbyObservern gehört z.B. auch die AbstractLobbyActivity und implementiert die Methode ***conncetionStateChanged(int state)***, welche aufgerufen wird, wenn der Service eine Verbindung aufbaut oder trennt. Der BusObserver hört auf den Bus selbst und definiert die folgenden Methoden:

```
1 public void foundAdvertisedName(String channelName);
2 public void lostAdvertisedName(String channelName);
3 public void busDisconnected();
```

Codeausschnitt 3.8: BusObserver

Dadurch kann man zusätzlich noch auf das Öffnen und Schließen der Spiele reagieren, falls die in der PTPLibrary vorimplementierte Behandlung, für den Entwickler nicht ausreichend ist. Die ***foundAdvertisedName*** Methode wird aufgerufen, wenn eine neue Session gefunden wurde und ***lostAdvertisedName*** wenn eine bekannte Session geschlossen wurde oder nicht mehr erreichbar ist. Die ***busDisconnected*** Methode wird aufgerufen, wenn das BusAttachement keine Verbindung zum Bus mehr hat. Der BusObserver ist z.B. im PTPHelper schon einmal implementiert und er sorgt dafür, dass neu erstellte Spiele in einer Liste ***foundChannels*** gespeichert werden und entsprechend auch entfernt. Dies ist notwendig um in der Lobby alle gerade offenen Spiele anzeigen zu können.
Der SessionObserver definiert Methoden um auf Events zu reagieren, die im Bezug zu einer Session stehen. Der Codeausschnitt 3.9 zeigt die Methoden, welche das Interface definiert.

```
1 public void memberJoined(String uniqueId);
2 public void memberLeft(String uniqueId);
3 public void sessionLost();
```

Codeausschnitt 3.9: SessionObserver

Dies ermöglicht es dem Entwickler bestimmte Aktionen auszuführen, falls neue Spieler sich zur Session verbinden ***(memberJoined)***, sich von der Session trennen ***(memberLeft)*** oder man selbst die Verbindung zur Session verliert ***(sessionLost)***. Schließlich gibt es noch den ***ServiceHelperObserver***, welcher z.B.vom PTPService implementiert wird. Dieser Observer hört auf den Helper, welcher über die Methode ***doAction(int action)*** verschieden Aktionen mitteilt, die über den Helper ausgeführt werden. So hat z.B. der Methodenaufruf aus Codeausschnitt 3.10

```
1 PTPHelper.getInstance().joinChannel();
```

Codeausschnitt 3.10: verbinde zur Session

zur Folge, dass der PTPHelper den Service über den Observer mitteilt sich zu einem Channel zu verbinden, was im Genaueren so aussieht der Codeausschnitt 3.11

```
1 public synchronized void joinChannel() {
2   notifyHelperObservers(PTPService.JOIN_SESSION);
3 }
```

Codeausschnitt 3.11: JoinSession

Die oben genannten Observer sind mithilfe eines ***Handlers*** vom Hintergrundthread abgekoppelt. Dies verhindert, dass der Entwickler bei der Implementierung der Observer sich kaum Gedanken um Threadkollisionen machen muss. Der PTPHelper besitzt nämlich einen ***BackgroundHandler***, welcher im Konstruktor erstellt und an den UI Thread gekoppelt wird, siehe Codeausschnitt 3.12.

```
1 backgroundHandler = new BackgroundHandler(Looper.getMainLooper());
```

Codeausschnitt 3.12: UI Thread BackgroundHandler

Sobald der Hintergrundprozess aus dem PTPService die Observer benachrichtigen will und z.B. die Methode ***notifyMemberJoined(String)*** ausführt, wird diese Nachricht an den ***BackgroundHandler*** geleitet, wie das funktioniert zeigt der Codeausschnitt 3.13.

```
1 Message obtainedMessage = backgroundHandler.obtainMessage(BackgroundHandler.
      MEMBER_JOINED, membersId);
2 backgroundHandler.sendMessage(obtainedMessage);
```

Codeausschnitt 3.13: Nachricht an den BackgroundHandler leiten

Der ***BackgroundHandler*** reagiert über die Methode ***handleMessage(Message)*** auf die Nachricht, indem jedoch die Methode auf dem UI Thread ausgeführt wird.

Dann gibt es noch das ***SessionJoinRule*** Interface, welches die ***boolean canJoin(String joinersUniqueId)*** Methode definiert. Darüber kann der Entwickler bestimmte Regeln festlegen, die darüber entscheiden ob ein Mitspieler sich zu einem Spiel verbinden kann oder nicht. Dabei werden beim Verbinden eines Spielers alle diese Regeln durchlaufen und nur wenn alle davon ***true*** zurückliefern, darf der Spieler beitreten.

3.2 LobbyActivity

Die LobbyActivity ist eine abstracte Klasse, die eine Grundfunktionalität mitbringt, sodass man sich darum nicht mehr selbst kümmern muss. Sie bietet eine UI-Oberfläche, welche es ermöglicht einen Namen für den Spieler anzugeben, ein Spiel zu eröffnen oder sich zu einem Spiel zu verbinden. Auf dem Bild sieht man die zwei Einträge ***game*** und ***game2*** bei dem es sich um zwei gerade offene Spiele handelt. Um sich zu denen zu verbinden muss man nur den Namen berühren und schon verbindet sich zu dem Spiel. Da es sich jedoch um eine abstrakte Klasse handelt, muss man nämlich die Methoden ***getJoinChannelView*** und ***getHostChannelView*** implementieren. Bei diesen Methoden handelt es sich um eine definition der Klassen, die die Activity beschreiben, welche geöffnet werden soll, wenn eine gegebene Aktion durchgeführt wird. Das heißt wenn ein neues Spiel erstellt wird, so muss die Methode ***getHostChannelView()*** die Activity Klasse definieren, welche geöffnet werden soll, wenn ein Spiel geöffnet wird. Es spricht natürlich auch nichts dagegen die beiden Methoden die selbe Acti-

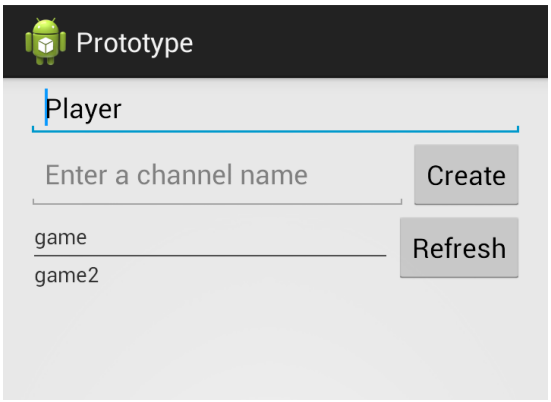


Abbildung 3.1: LobbyActivity

vity übergeben zu lassen, solange die Activity die logische Auseinandersetzung mit dem Host und Clienten übernimmt. Die Behandlung der Buttons ***create*** und ***refresh*** werden durch die Methoden ***refresh(View)*** und ***create(View)*** representiert. Die Methode ***create*** z.B. überprüft zuerst ob der Spielername und der Spielname angegeben sind und startet darauf die Session, wie im Codeausschnitt 3.14 aufgeführt.

```
1 PTPHelper.getInstance().setHostChannelName(channelName);
2 PTPHelper.getInstance().setPlayerName(playerName);
3 PTPHelper.getInstance().hostStartChannel();
4 updateUIState(PTPHelper.SESSION_HOSTED);
```

Codeausschnitt 3.14: Session starte

Der letzte Befehl graut alle UI Elemente aus, sodass während man verbindet keine weiteren Aktionen ausgeführt werden können. Das Verbinden zu einen Spiel geschieht ähnlich, jedoch über einen ***ClickListener***, da es sich bei den Einträgen um eine Liste handelt, welche bei der Berührung den ***ClickListener*** entsprechend aufrufen. Natürlich kann der Entwickler auch seine eigene Lobby implementieren, dazu muss er nur die Klasse ***AbstractLobbyActivity*** implementieren, welche nur die definition der Methode ***connectionStateChanged(int)*** aus dem LobbyObserver besitzt und vom PTPHelper aufgerufen wird, wenn sich der Verbindungszustand ändert.

3.3 PTPService

Die PTPService Klasse besitzt die eigentliche Funktionalität der Bibliothek und ist dafür verantwortlich die Verbindung mit dem Bus zu erstellen und entsprechend zu konfigurieren. Der PTPService ist eine Unterklasse von Service aus dem Android SDK, welche im Hintergrund gestartet wird und in einem seperaten Thread läuft. Der Service wird von der Android VM selber gestartet und es gibt keine direkten Zugriff auf die Instanz von dieser Service Klasse, somit ist man darauf angewiesen die Kommunikation mit einem Nachrichtensystem zu realisieren. Android bietet hierfür ein Nachrichtensystem an um mit dem Service zu kommunizieren. Jedoch wurde in diesem Rahmenwerk auf die Verwendung dessen, bis auf den ***Handler***, weitgehend verzichtet um zum einen eine gewissen Unabhängigkeit vom Android SDK zu bekommen, denn wie schon erwähnt ändert sich die API des Android SDKs gelegentlich. Weiterhin ist die Verwendung des weitbekannten Observer-Patterns eine gute Wahl um eine Flexibilität bei der Weiterentwicklung des PTPFramework zu

gewährleisten. Wenn der PTPService erstellt wird, wird seine ***onCreate()*** Methode aufgerufen in der der Service sich als Observer beim PTPHelper registriert. Daraufhin kann der PTPService Nachrichten erhalten um Aktionen, wie sich zu einem Spiel zu verbinden oder Eins zu erstellen, ausführen. Damit das Bussystem von AllJoyn initialisiert wird, bevor man damit interagieren kann, muss man den folgenden Befehl ausführen:

```
1 org.alljoyn.bus.alljoyn.DaemonInit.PrepareDaemon(getApplicationContext());
```

Codeausschnitt 3.15: Daemon initialisieren

Der AllJoyn Daemon ist ein Hintergrundprozess, welcher sich um die verbindungstechnischen Abläufe kümmert. Daraufhin wird der HandlerThread gestartet, welcher sich um die Nachrichten von PTPHelper kümmert. Dabei wird wie beim PTPHelper ein BackgroundHandler initialisiert, jedoch mit einem neu erstellem Thread. Der Codeauschnitt 3.16

```
1    public void exit() {}
2  public void connect() { }
3  public void disconnect() {}
4  public void startDiscovery() {}
5  public void cancelDiscovery() {}
6  public void requestName() {}
7  public void releaseName() {}
8  public void bindSession() {}
9  public void unbindSession() {}
10 public void advertise() {}
11 public void cancelAdvertise() {}
12 public void joinSession() {}
13 public void leaveSession() {}
```

Codeausschnitt 3.16: Methoden des BackgroundHandlers

Jede dieser Methoden tut im Grunde nichts Anderes als den Befehl in eine Nachricht zu packen und an den anderen Thread zu schicken, welcher zuvor bei der Methode ***startBusThread()*** erstellt wurde. Die Methoden vom BackgroundHandler werden wiederum über die Methode ***doAction(int)*** ausgeführt, welche durch den HelperObserver definiert ist. Dabei wird das Argument über einen Switchcase ausgewertet und daraufhin die entsprechende Methode ausgeführt.

In den nächsten Unterkapiteln wird auf die einzelnen Methoden eingegangen, welche sich im PTPService befinden und vom BackgroundHandler ausgeführt werden.

3.3.1 doConnect() und doDisconnect()

Die ***doConnect()*** Methode ist dafür verantwortlich eine Verbindung mit dem Bus herzustellen indem es ein BusAttachement erstellt, den SignalHandler und das BusObject registriert. Der Codeauschnitt 3.17 zeigt wie das BusAttachement erstellt wird.

```
1 bus = new BusAttachment(package + "appName",BusAttachment.RemoteMessage.Receive);
```

Codeausschnitt 3.17: BusAttachement erstellen

Dabei wird als erster Parameter der BusAttachement Name übergeben. Der Name ist dafür notwendig mehrere BusAttachement bestimmten Anwendungen zuzuordnet, da die Kommunikation über den selben Daemon läuft. Dies verhindert z.B. dass die Kommunikation von zwei unterschiedlichen Spielen, welche beide über das PTPFramework oder direkt mit AllJoyn realisiert sind, sich in die Quere kommen. Der zweite Parameter bestimmt ob eingehende Nachrichten empfangen und behandelt oder einfach verworfen werden sollen. Danach wird in der Methode der BusListener registriert, welcher dafür zuständig ist auf geöffnete Spiele zu reagieren. Diese Nachrichten werden dann an die Observer weiter geschickt, siehe Codeausschnitt 3.18

```
1  @Override
2  public void foundAdvertisedName(String fullName, short transport, String
       namePrefix) {
3    if(namePrefix.equals(packageName))
4      PTPHelper.getInstance().notifyFoundAdvertisedName(getSimpleName(fullName));
5  };
6  @Override
7  public void lostAdvertisedName(String fullName, short transport, String
       namePrefix) {
8    if(namePrefix.equals(packageName))
9      PTPHelper.getInstance().notifyLostAdvertisedName(getSimpleName(fullName));
10 }
11 @Override
12 public void busDisconnected() {
13   PTPHelper.getInstance().notifyBusDisconnected();
14 }
```

Codeausschnitt 3.18: BusListener Nachrichten

Die Methode ***getSimpleName(String)*** tut nichts anderes, als aus einem vollständigen Namen, welcher sich aus dem Packagenamen und dem Channelnamen zusammensetzt, nur den Channelname zu extrahieren und an die Observer weiterzuschicken. Da der Packagename sich innerhalb der Anwendung nicht ändert, wird auch nur der Channelname verwendet. Daraufhin wird der BusObject registriert, was im Codeausschnitt 3.19 zu sehen ist.

```
1 Status status;
2 BusObject busObject = PTPHelper.getInstance().getBusObject();
3 if(busObject != null){
4   status = bus.registerBusObject(busObject, objectPath+ "/" + getDeviceID());
5   if (Status.OK != status) {
6     Log.e(TAG, "Cannot register : " + status);
7     return;
8   }
9 }
```

Codeausschnitt 3.19: BusObject registrieren

Dabei wird der BusObject genommen welcher bei der Initialisierung vom PTPHelper übergeben wurde. Der BusObject wird später dazu benötigt einen sogenannte SignalEmmiter zu bekommen, über welchen man Nachrichten an andere Teilnehmer schicken kann. Der Pfad, unter welchen der BusObject abgelegt wird, muss innerhalb des BusSystems einzigartig sein, um Konflikte zwischen den BusObjekten zu vermeiden. Wie man an dem Beispiel sieht, ist der Pfad wie bei einen normalen Dateisystem organisiert und wird über das Slashzeichen getrennt. Um die Einzigartigkeit jedes Bus Objektes zu ermöglichen wurde als Teilpfad die Geräte ID benutzt. Wie man an die Geräte ID kommt, zeigt der Codeausschnitt 3.20.

```
1 private String getDeviceID(){
2   TelephonyManager telephonyManager = (TelephonyManager) getBaseContext().
        getSystemService(Context.TELEPHONY_SERVICE);
3   return telephonyManager.getDeviceId();
4 }
```

Codeausschnitt 3.20: Geräte ID

Jedoch muss man dazu noch in dem AndroidManifest die notwendige Erlaubniss ***READ_PHONE_STATE*** hinzufügen. Daraufhin wird die ***bus.connect()*** Methode aufgerufen um mit dem Bus zu verbinden. Wenn die Verbindung erfolgreich war, was man über den Status als Rückgabewert erfährt, wird der SignalHandler registriert, welcher zuvor bei der Initialisierung des PTPHelper übergeben wurde.

Schließlich wird die UniqueID des BusAttachements an den PTPHelper übergeben, welcher einen busweiten einzigartigen String darstellt, sowie auch der Zustand der Verbindung auf ***CONNECTED*** gesetzt, siehe Codeausschnitt 3.21

```
1 PTPHelper.getInstance().setUniqueID(bus.getUniqueName());
2 PTPHelper.getInstance().setConnectionState(PTPHelper.CONNECTED);
```

Codeausschnitt 3.21: UniqueID und Verbindungstatus setzen

Die ***doDisconnect()*** Methode ist im Prinzip dafür verantworlich alle Handler und das BusObject abzumelden und die Verbindung zu trennen.

3.3.2 doStartDiscovery() und doStopDiscovery()

Die Methode ***doStartDiscovery()*** wird meist zu begin, nach dem die Verbindung aufgebaut ist, ausgeführt. Die ist dafür verantworlich dem BusAttachement mitzuteilen auf offene Spiele zu horchen und die dem Observer mitzuteilen. Dies geschieht über den einfachen Methodenaufruf ***bus.findAdvertisedName(packageName)***. Wobei der ***packageName*** das Prefix für den Name darstellt. Somit würde das BusAttachement auf alle offene Spiele reagieren, welche den Prefix mit dem ***packageName*** haben. Um dann das Entdecken von offenen Spielen auch zu beenden kann wird die Methode ***doStopDiscovery()*** ausgeführt, welche dann die Methode ***bus.cancelFindAdvertisedName(packageName)*** ausführt.

3.3.3 doBindSession() und doUnbindSession()

Die ***doBindSession()*** Methode wird dafür benötigt um eine neue Session zu erstellen, wenn man ein neues Spiel erstellt. Dabei werden unterschiedliche Konfigurationen vorgenommen. Zum einen muss man den SessionPort festlegen und die Transportparameter konfigurieren. Bei den Transportparametern handelt es sich darum, ob man die Verbindung über Bluetooth, WifiDirect oder Lan zulässt, ob es sich hierbei um eine Multipoint Session handelt, ob die Session Beitreter von außerhalb des lokalen Gerätes befinden dürfen und ob die Daten roh oder als Nachrichten verpackt verschickt werden. In diesem Fall wird nur die Verbindung über Wlan zugelassen, es soll sich um eine Multipointverbindung handeln, die Daten sollen als Nachrichten verschickt werden und es dürfen alle sich zu der Session verbinden, also auch von anderen Geräten aus. Wie die Konfiguration vorgenommen wird sieht man im Codeausschnitt 3.22

```
Mutable.ShortValue mutableContactPort = new Mutable.ShortValue(contactPort);
SessionOpts sessionOpts = new SessionOpts(SessionOpts.TRAFFIC_MESSAGES,
       true, SessionOpts.PROXIMITY_ANY, SessionOpts.TRANSPORT_WLAN);
```

Codeausschnitt 3.22: Sessionkonfiguration

Als ***contactPort*** wird der Standartwert 100 genommen. Neben den Konfigurationsparametern benötigt man auch einen ***SessionPortListener***, welcher sich um die Verbindungsanfragen von anderen Geräten kümmert, sowie bei einer erfolgreichen Verbindung die nötigen Schritte einleitet. Der ***SessionPortListener*** implementiert eine Methode, die im Codeausschnitt 3.23:

```
public boolean acceptSessionJoiner(short
sessionPort, String joiner, SessionOpts sessionOpts) {
    if(sessionPort == contactPort){
      return PTPHelper.getInstance().canJoin(joiner);
    }
    return false;
}
```

Codeausschnitt 3.23: SessionPortListener implementierung

Dabei wird zuerst geprüft ob der SessionPort übereinstimmt und danach werden die möglicherweise definierten ***SessionJoinRules*** durchlaufen. Erst wenn alle die Regeln ***true*** zurückliefern wird das Beitreten zur Session gewährt.

```
public void sessionJoined(short sessionPort,int sessionId,String joiner){
if(firstJoiner){
  firstJoiner = false;
  hostSessionId = sessionId;
  SignalEmitter emitter = new SignalEmitter(PTPHelper.getInstance().getBusObject
      (), sessionId, SignalEmitter.GlobalBroadcast.Off);
  hostInterface = emitter.getInterface(PTPHelper.getInstance().
      getBusObjectInterfaceType());
  PTPHelper.getInstance().setSignalEmiter(hostInterface);
  PTPHelper.getInstance().notifyMemberJoined(joiner);
  bus.setSessionListener(sessionId, new PTPSessionListener());
  }
}
```

Codeausschnitt 3.24: Als Host den SignalEmitter bekommen

Die zweite Methode umfasst ein etwas kompliziertes Szenario. AllJoyn erlaubt es nicht einen Host, welcher eine Session bindet, sich zu dieser auch zu verbinden.

Um jedoch als Host auch am Spiel teilnehmen zu können wird dieses Verhalten durch einen Umweg erreicht. Durch die Methode aus dem Codeausschnitt 3.24 erfährt man die bis dahin unbekannte ***SessionID***, da sie durch das Binden der Session nicht mitgeteilt wird. Daraufhin kann man mit der ***SessionID*** einen ***SignalEmitter*** erstellen, der als Endpunkt fungiert und das Verschicken von Nachrichten ermöglicht. Zusätzlich lässt sich der ***PTPSessionPortListener*** an die Session binden. Somit ist es dem Host nur möglich zu funktionieren, wenn sich der erste Teilnehmer verbunden hat.Zum schluss wird der ***SessionPortListener*** beim Bus registriert und eine Session erstellt, siehe Codeausschnitt 3.25.

```
1 bus.bindSessionPort(mutableContactPort, sessionOpts, sessionPortListener);
```

Codeausschnitt 3.25: Session erstellen

In der Methode ***doUnbindSession()*** wird durch den einfachen Methodenaufruf ***bus.unbindSessionPort(contactPort)***, die Session, welche an den ***contactPort*** gebunden ist, geschlossen:

3.3.4 doRequestName() und doReleaseName()

Die Methoden ***doRequestName()*** und ***doReleaseName()*** sind dazu da um die Spielnamen, welche man dann später durch das ***Advertising*** an andere Teilnehmer mitteilen kann, vom Bus zugewiesen zu bekommen. Man kann nämlich nur Namen mitteilen, welche man auch zugewiesen bekommen hat, denn es kann ja vorkommen, dass ein Name schon vergeben ist. Der Codeausschnitt 3.26 zeigt die genaue Implementierung.

```
1 bus.requestName(packageName + "." +PTPHelper.getInstance().getHostChannelName(),
2         BusAttachment.ALLJOYN_REQUESTNAME_FLAG_DO_NOT_QUEUE);
```

Codeausschnitt 3.26: Name für Session anfordern

Somit wird als Prefix der ***packgeName*** verwendet und der ***HostChannelName*** ist der Name, welche in der Lobby hinterlegt und beim PTPHelper abgespeichert wurde. Der zweite Parameter besagt, dass bei vorhanden Namen nichts unternommen werden soll. Eine andere Möglichkeit bestehe nämlich den vorhandenen Namen einfach zu ersetzen. Durch ***doReleaseName()*** wird der Name dann wieder freigegeben, sodass er wieder von Anderen verwendet werden kann.

3.3.5 doAdvertise() und doCancelAdvertise()

Das sogenannte ***Advertising*** wird dazu benutzt um anderen Teilnehmern den Spielnamen mitzuteilen. Somit wird über einen einfachen Befehl die Mitteilung gestartet und beendet. Zur Verdeutlichung siehe Codeausschnitt 3.27.

```
1 String wellknownName =packageName+"."+PTPHelper.getInstance().getHostChannelName
      ();
2 bus.advertiseName(wellknownName, SessionOpts.TRANSPORT_WLAN);
3 ...
4 String wellknownName = packageName+"."+PTPHelper.getInstance().getHostChannelName
      ();
5 bus.cancelAdvertiseName(wellknownName,SessionOpts.TRANSPORT_ANY);
```

Codeausschnitt 3.27: Session Name mitteilen

Dabei werden zum einen der Name des Spiels übergeben, als auch das Transportmedium, in diesem fall Wlan.

3.3.6 doJoinSession() und doLeaveSession()

Die Methode ***doJoinSession()*** ist sogesehen das entsprechende Gegenbeispiel zu der ***doBindSession()*** Methode. Diese Methode wird von den beitretenden Teilnehmer ausgeführt. Im Codeausschnitt 3.28 sieht man die genauere Vorgehensweise beim Bestreten einer Session.

```
1 String wellKnownName = packageName + "." + PTPHelper.getInstance().getChannelName
      ();
2 SessionOpts sessionOpts = new SessionOpts(SessionOpts.TRAFFIC_MESSAGES,
3     true, SessionOpts.PROXIMITY_ANY, SessionOpts.TRANSPORT_WLAN);
4
5 Mutable.IntegerValue sessionId = new Mutable.IntegerValue();
6
7 bus.joinSession(wellKnownName, contactPort, sessionId,sessionOpts,
      sessionListener);
```

Codeausschnitt 3.28: Session beitreten

Hierbei werden über die /ibSessionOpts wie bei der ***doBindSession()*** Methode die ganzen Parameter für die Session festgelegt. Bei der ***sessionId*** handelt es sich um einen veränderbaren integer Wert, welcher bei der Ausführung der Methode zugewiesen wird. Schließlich wird noch ein ***SessionListener*** als Parameter übergeben.

Auch dieser Listener leitet die Nachrichten an die Observer, welche im PTPHelper abgespeichert sind, weiter. Jedoch wird die Information über die SessionId dem PTPHelper nicht mitgeteilt, da sie für den Entwickler, welcher den Observer gegebenenfalls implementiert, nicht von Interesse sein sollte. Falls die Verbindung erfolgreich war wird anschließend ein ***SignalEmitter*** erstellt und dieser an den PTPHelper übergeben, wie im Codeausschnitt 3.29 aufgeführt.

```
1 SignalEmitter emitter = new SignalEmitter(PTPHelper.getInstance().getBusObject(),
      clientSessionId, SignalEmitter.GlobalBroadcast.Off); Object clientInterface
     = emitter.getInterface(PTPHelper.getInstance().getBusObjectInterfaceType());
     PTPHelper.getInstance().setSignalEmitter(clientInterface);
2 PTPHelper.getInstance().setConnectionState(PTPHelper.SESSION_JOINED);
```

Codeausschnitt 3.29: SignalEmitter bekommen als Client

3.3.7 doQuit()

Schließlich gibt es noch die ***doQuit()*** Methode, die aufgerufen wird, den der Service beendet werden soll. Der Codeausschnitt 3.30 zeigt den Inhalt dieser Methode.

```
1 backgroundHandler.disconnect();
2 backgroundHandler.getLooper().quit();
3 PTPHelper.getInstance().removeObserver(this);
4 this.stopSelf();
```

Codeausschnitt 3.30: doQuit Methode

Es wird somit die Verbindung mit dem Bus geschlossen, falls sie noch offen ist. Dann wird der ***BusThread*** beendet, welcher an den ***BackgroundHandler*** gekoppelt war. Und zum Schluss wird der Service als Observer vom PTPHelper entfernt und beendet.

4 Praktisches Anwendungsbeispiel: Graphenspiel

4.1 Spielidee

Das Prinzip des Graphenspiel basiert im Grunde darauf, dass die Spieler einen Graph vorgelegt bekommen, bei dem sich die Kanten überschneiden. Ziel des Spieles ist es die Knoten des Graphen so zu bewegen, dass sich keine Kante mehr überschneidet. Der Hacken hinter der Sache ist jedoch, dass sich alle anderen Knoten mitbewegen, falls einer bewegt wird, bis auf die, die von den anderen Spielern gehalten bzw. bewegt werden. Genau darin liegt der Mehrspielergedanke, denn man kann das Spiel nicht alleine gewinnen. Somit handelt es sich hierbei um ein sogenanntes kooperatives Mehrspielerspiel. Auf dem Bild 4.1 sieht man z.B. dass der Knoten mit der Nummer 6 rot markiert ist. Somit kann der Spieler diesen Knoten nicht bewegen. Aber er kann alle anderen Knoten, welche blau markiert sind, bewegen. Das Puzzel lässt sich einfacher lösen, wenn mehr Leute mitspielen, was auch einen positiven sozialen Aspekt mit sich bringt. Außerdem ist es sehr von Vorteil bei der Lösung des Puzzels sich mit den anderen Mitspielern abzusprechen, sodass man auf ein gemeinsames Ziel hinarbeiten kann. Zum Schluss wird noch die Zeit angezeigt, die gebraucht wurde um das Rätsel zu lösen, was dazu motivieren kann es nochmal zu versuchen und es schneller zu schaffen. Weiterhin lässt sich wärend des Spielens zu dem Spiel ver-

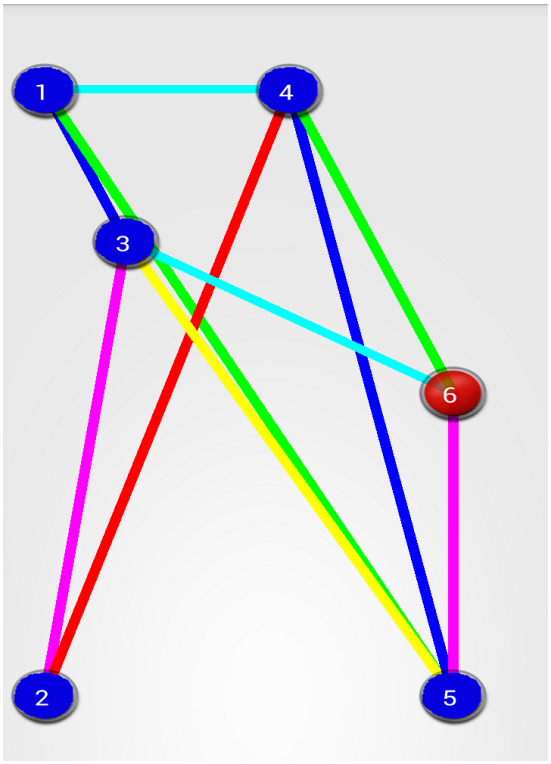


Abbildung 4.1: Graphspiel

binden, d.h. selbst wenn ein Spiel schon im Gange ist, kann ein weiterer Spieler dazu stoßen und mithelfen. Natürlich kann somit auch jeder das Spiel verlassen, ohne das laufende Spiel beenden zu müssen, unter Beachtung, dass man immer noch mindestens zwei Mitspieler braucht um das Rätsel lösen zu können.

4.2 Anforderungen

Hier nochmal die Auflistung der Anforderungen, welche zur Implementierung des oben genannten Spiels notwendigt sind, also in der Fachsprache auch Must-Haves genannt.

- Es soll zu Begin ein Graph erstellt werden mit mindestens 6 Knoten
- Zu Begin müssen sich die Knoten überschneiden
- Als Spieler kann man alle Knoten verschieben, die von keinem anderen Spieler im Moment verschoben werden.
- Als Spieler sieht man in Echtzeit wie sich eigene und die von den anderen Mitspielern verschobene Knoten bewegen.
- Als Spieler sollte man sofort sehen, wenn man gewonnen hat.
- Als Spieler kann ich einen Namen angeben, welchen von anderen Mitspielern gesehen wird.
- Der generierte Graph soll immer lösbar sein
- Am Ende des Spiels, sollte die Zeit angezeigt werden, die für das Lösen des Rätsels benötigt wurde.

4.3 Implementierung

4.3.1 Projektkonfiguration

Für die Entwicklung des Spieles wurde weiterhin Eclipse verwendet, sodass die Beschreibung der Projektkonfiguration sich auf Eclipse beruhen. Als Erstes wurde ein ***Android Application Project*** erstellt. Dabei wurden die Applicationname, Packagename und Projektname entsprechend angegeben. Die restlichen Einstellung wurden als Standarteinstellung belassen, bis auf die ***Activity***, welche ausgelassen wurde, da für die Implementierung eine eigene Application Klasse geschrieben wird. Weiterhin wurde als Target SDK ***Android 4.2 Jelly Bean*** und als Mininum ***Android 2.3*** verwendet, da AllJoyn mindestens die Android Version 2.3 benötigt. Als nächstes ist es notwendig das ***PTPLibrary*** Projekt als Bibliothek im Spielprojekt zu referenzieren wie im Bild 4.2 zu sehen. Dazu klickt man rechts auf das Projekt und dann auf Eigenschaften. Unter den Reiter ***Android*** gibt es unten die Einstellung für Bibliotheken. Hier muss man die ***PTPLibrary*** als Bibliothek angeben. Leider erlaubt es Android nicht ***Libraries*** als gewöhnliche Jar Dateien zu packen, daher muss man die Bibliothek als Projekt referenzieren.[6]

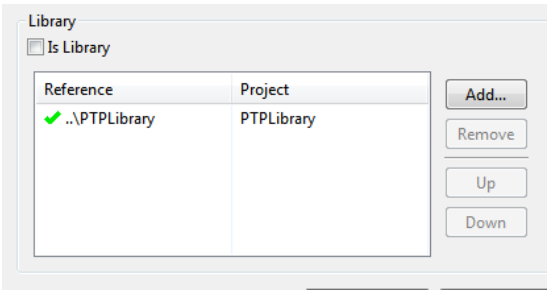


Abbildung 4.2: Bibliothekreferenz

4.3.2 MainApplication

Die ***MainApplication*** Klasse wird beim Ausführen der Anwendung als erstes ausgeführt und ist für die Initialisierung von für das Spiel wichtigen Objekten verantwortlich. Am Anfang die die ***onCreate()*** Methode aufgerufen. Im Codeausschnitt 4.1 sieht man den genaueren Inhalt der Methode.

```
1 super.onCreate();
2 graph = new Graph();
3 graph.addObserver(this);
4 graph.setupPoints();
5 PTPHelper.initHelper("GraphGame",GraphInterface.class, this, new GraphDummyObject
      (), graph, GraphLobbyActivity.class);
```

Codeausschnitt 4.1: onCreate Methode

Dabei wird als erstes das Graph Objekt erstellt und konfiguriert, welches das Datenmodel des Spieles representiert. Als zweites wird dann der ***PTPService*** initialisiert, mit der Angabe der nötigen Parameter. Interessant ist hier das ***GraphDammyObject***, welches das BusObject respresentiert. Es implementiert das ***GraphInterface*** und das ***BusObject*** Interface, beinhaltet aber keine weitere Logik, da es für das Spiel irrelevant ist und nur für AllJoyn benötigt wird für eine erfolgreiche Registrierung des ***BusAttachements***. Das ***graph*** Object spielt die Rolle des SignalHandlers und implementiert die Methoden des BusIntefaces. Die ***MainApplication*** implementiert außerdem noch den ***GraphObserver***, welcher die Methode ***doAction(int)*** definiert. Darüber kann der Graph der ***MainApplication*** mitteilen, Nachrichten an andere Teilnehmer zu schicken. Dies wird zuerst an den ***MessageHandler*** weitergeleitet, welcher über die die Nachricht genauer behandelt, wie im Codeausschnitt 4.2 sichtbar.

```
1  public void handleMessage(Message msg) {
2    try{
3      GraphInterface remoteGraph = (GraphInterface) PTPHelper.getInstance().
           getSignalEmitter();
4      switch (msg.what) {
5      case Graph.NODE_POSITION_CHANGED:
6        Node node;
7        while(( node = graph.getChangedNode()) != null){
8          remoteGraph.MoveNode(node.getId(), node.x, node.y, PTPHelper.getInstance()
               .getUniqueID());
9        }break;
10
11     case Graph.POINT_OWNERSHIP_CHANGED:
12       Graph.IdChange idChange;
13       while(( idChange = graph.getIdChange()) != null){
14         remoteGraph.ChangeOwnerOfNode(idChange.id, idChange.owner, PTPHelper.
               getInstance().getUniqueID());
15       }break;
16
17     default: break;
18     }
19   }catch(BusException e){
20     Log.e(TAG, "BusException: " + e);
21   };
22 }
```

Codeausschnitt 4.2: Graphnachrichten behandeln

Zu erst wird über einen Statuscode der Nachrichtentyp mitgeteilt, worauf der ***MessageHandler*** sich die nötigen Information, dann vom Graphobject holt und über den ***SignalEmitter*** an die anderen Teilnehmer schickt. Man sieht, dass die Kommunikation recht einfach gehalten ist, denn es gibt nur zwei Nachrichtentypen, nämlich wenn jemand den Knoten berührt und wenn jemand den Knoten verschiebt. Der rest wird durch die Logik bei den einzelnen Teilnehmern entschieden. Gerade bei Echtzeitspielen ist es wichtig die Nachrichten, so gering wie möglich zu halten.

4.3.3 Graph

Der ***Graph*** representiert das Datenmodel des Spiels und beinhaltet somit alle Spielrelevanten Daten. Die Beschreibung des Graphen wird durch einfache Kanten und Knoten gespeichert, welche in einer Liste abgelegt werden. Jede Kante verweist auf zwei Knoten und jeder Knoten besitzt eine ***ID*** und eine Positionn als X- und Y-Koordinate. Der ***Graph*** implementiert das ***GraphInterface***, welches das BusObject-Interface für die Kommunikation darstellt. Die Methoden des Interfaces sind im Codeausschnitt 4.3 angeben.

```
1 @BusSignal
2 public void MoveNode(int id,double x,double y, String uniqueName) throws
      BusException;
3 @BusSignal
4 public void ChangeOwnerOfNode(int id,String owner, String uniqueID) throws
      BusException;
```

Codeausschnitt 4.3: GraphInteface

AllJoyn schreibt es vor bei den Methoden, welche für die Kommunikation über den Bus genutzt werden,mit einer entsprechenden Annotation zu erweitern, sodass die Nachrichten entsprechend gehandhabt werden können. In diesem Fall handelt es sich um ***Signalmethoden***, d.h. diese Methoden werden dazu benutzt um Nachrichten an alle Teilnehmer gleichzeitig zu schicken. Die konkrete Implementierung dieser Methoden beim ***SignalHandler*** ist somit die Behandlung der eingehenden Nachrichten. Die Annotationen gehören zum einen in das Interface als auch in die konkrete Implementierung. Die Implementierung der ***MoveNode*** Methode sieht man im Codeausschnitt 4.4.

```
@BusSignalHandler(iface = "com.example.firstapp.GraphInterface", signal = "
    MoveNode")
public synchronized void MoveNode(int id, double x, double y, String uniqueName)
    throws BusException {

  for (Node point : nodes) {
    if(point.getOwner().isEmpty() || point.getOwner().equals(uniqueName)){
      point.x += x;
      point.y += y;
      checkAndAdjust(point);
    }
  }
  if(uniqueName.equals(PTPHelper.getInstance().getUniqueID())){
    Node changedNode = new Node(x,y,id);
    changedNodes.add(changedNode);
    notifyObservers(NODE_POSITION_CHANGED);
    return;
  }
  notifyObservers(GRAPH_CHANGED);
}
```

Codeausschnitt 4.4: MoveNode implementierung

Wichtig hier ist auch die richtige Annotation der Methode, welche zum einen das Inteface nochmal angibt und den Methodennamem. Die Method selbst prüft erst ob der zu verschiebende Knoten auch dem Spieler gehört, wenn ja wird der Offset der als Parameter hereingereicht wird hinzuaddiert. Anschließend werden die Knoten, welche aus dem Bildschirm herausgehen, durch die Methode ***checkAndAdjust()*** zurück an die Bildschirmgrenze gebracht, sodass die Knoten den Bildschirm nicht verlassen können. Als zweiter Schritt wird geprüft ob der Spieler der den Knoten verschoben hat, auch der Spieler ist welcher lokal auf dieser Instanz des Spiels spielt. Wenn es der Fall ist, werden die Änderungen als Knoten in eine Liste abgespeichert und den Observer wird gesagt, dass er die Änderung den anderen Mitspielern mitteilen kann. Zum Schluss wird den Observern noch mitgeteilt, dass ich der Graph verändert hat, sodass z.B. die View den Graphen neu zeichnen kann. Die zweite Methode ***ChangeOwnerOfNode()*** kümmert sich um das Zuweisen von Knoten zu den Spielern und ist im Codeausschnitt 4.5 zu sehen.

```
1  @BusSignalHandler(iface = "com.example.firstapp.GraphInterface", signal = "
       ChangeOwnerOfNode")
2  public synchronized void ChangeOwnerOfNode(int id, String owner, String uniqueID)
        throws BusException {
3    for (Node node : nodes) {
4      if(node.getId() == id){
5        node.setOwner(owner);
6        if(uniqueID.equals(PTPHelper.getInstance().getUniqueID())){
7          addIdOfChangedPoint(new IdChange(id,owner));
8          notifyObservers(POINT_OWNERSHIP_CHANGED);
9          return;
10       }
11       notifyObservers(GRAPH_CHANGED);
12       return;
13     }
14   }
15 }
```

Codeausschnitt 4.5: ChangeOwnerOfNode implementierung

Als Parameter wird erst die Knoten ID übergeben, damit man weiß um welchen Knoten es sich handelt. Dazu kommt erst der neue Besitzer und dann die ID des Spielers, von wem diese Nachricht stamm. Der letzte Parameter ist daher wichtig, falls die Besitzer ID leer ist, was soviel bedeutet, dass der Spieler den Knoten losgelassen hat, man noch nachvollziehen von wem die Nachricht stamm um zu entscheiden ob man den anderen Mitspielern es mitteilen soll oder nicht. Die Methode macht im Grunde auch nichts Anderes als den neuen Besitzer zu setzen und zu entscheiden ob dieser Wechsel mitgeteilt werden soll. Die Idee ist die folgende. Wenn man selber den Knoten losgelassen hat, müssen es die Anderen wissen, falls es jemand anders getan hat, so hat derjenige selber dafür gesorgt es jedem mitzuteilen.

4.3.4 DrawView

Schließlich gibt es noch die ***DrawView***, welche für die grafische Darstellung und die Benutzereingaben zuständig ist. Diese Klasse ist die Unterklasse von ***View*** und implementiert weiterhin den ***OnTouchListener*** und den ***GraphObserver***. Die ***View*** ist eine Android Klasse ist für die grafische Darstellung zuständig und beinhaltet als wichtigste Methode ***onDraw(Canvas)***, welche dafür zuständig ist die Kanten und Knoten auf dem ***Canvas*** zu "malen". Außerdem wird in dieser Methode geprüft ob der Graph gelöst ist und eine Meldung dass man gewonnen hat erscheinen

soll. Die Methode ***isGraphFinished()*** überprüft, durch die Benutzung eines Algorithmus [7], ob sich noch Kanten überschneiden. Durch die Implementierung des ***OnTouchListeners*** wird die Methode ***onTouch(View, MotionEvent)*** benutzt, welche sich um die Benutzeraktivitäten kümmert, wie das Bewegen von Knoten. Dabei wird z.B. bei der ersten Berührung geprüft ob der Knoten noch keinen Besitzer hat, und daraufhin der neue Besitzer zugewiesen. Die Abfrage ob es sich um die erste Berührung handelt, wird im Codeausschnitt 4.6 gezeigt.

```
1 if(event.getAction() == MotionEvent.ACTION_DOWN)
```

Codeausschnitt 4.6: Erste Berührung

Weiterhin gibt es noch den ***ACTION_ UP*** Bewegungstatus, welches das lösen des Fingers bedeutet. In diesem Fall wird der Knoten, falls einer diesem Spieler zugewiesen war, wieder freigegeben. Und schließlich gibt es noch den Fall, dass keiner dieser Fälle gegeben ist und dies bedeutet nur das der Finger bewegt wird. Hierbei wird die letzte Position des Knoten mit der momentanen Position des Fingers berechnet und die Differenz zum Knoten hinzuaddiert, also der Knoten wird auf die Position des Fingers bewegt. In jedem dieser Fälle wird die Änderung an die anderen Spieler mitgeteilt. Dann gibt es noch die Methode ***update(int)***des ***GraphObservers***, welche auf die Nachricht hört, dass der Graph neu gezeichnet werden soll, was man im Codeausschnitt **??** nochmal sieht.

```
1 @Override
2 public void update(int args) {
3   if(Graph.GRAPH_CHANGED == args)
4     postInvalidate();
5 }
```

Codeausschnitt 4.7: Graphen neu zeichnen

Die ***postInvalidate()*** Methode gehört zu der View Klasse und muss aufgerufen werden um dem UI Thread mitzuteilen, dass die View nicht aktuell ist. Falls man sich innerhalb des UI Threads befindet, wie z.B. bei der Abarbeitung der Berührungsaktivität reicht es die Methode ***invalidate()*** aufzurufen um das Neuzeichnen anzustoßen.

5 Praktisches Anwendungsbeispiel: MauMau

5.1 Spielidee

Das Spiel MauMau ist ein in Deutschland recht bekanntes Kartenspiel. Ziel dieses Kartenspiels ist es, alle seine Karten loszuwerden. Zusätzlich gibt es einige Regeln die man beachten muss. Um dieses Spiel spielen zu können benötigt man mindestens 2 Mitspieler und maximal 4 Spieler, da man bei dieser Version ein Skatblatt benutzt, welches nur 32 Karten umfasst. Am Anfang des Spiels, wenn alle sich verbunden haben bekommt jeder 6 zufällige Karten ausgeteilt und der Host,in dieser Version, darf anfangen. Dabei kann man pro Zug, wenn man dran ist eine Karte spielen. Falls noch keine Karte gespielt wurde, darf jede Karte gespielt werden. Daraufhin ist der nächste Spieler dran und muss eine den Regeln entsprechende Karte legen. Gewonnen hat somit der Spieler, welcher als Erstes alle seine Karten losgeworden ist. Das Bild 5.1 zeigt eine Bildschirmaufname von einer Spielpartie MauMau. Dabei sieht man im oberen Bereich eine Liste mit allen Teilnehmenden Spielern und dahin die Anzahl der Karten, die jeder dieser Spieler besitzt. In der Mitte sieht man die zuletzt gespielte Karte, welche einen König Karo darstellt. Die Karten ganz unten im Bild sind ein Auschnitt der Karten, die der Spieler momentan besitzt. Die eigenen Karten sind als eine Bildergallerie dargestellt und lassen sich somit durch eine gewöhnliche Fingebewegung scrollen.

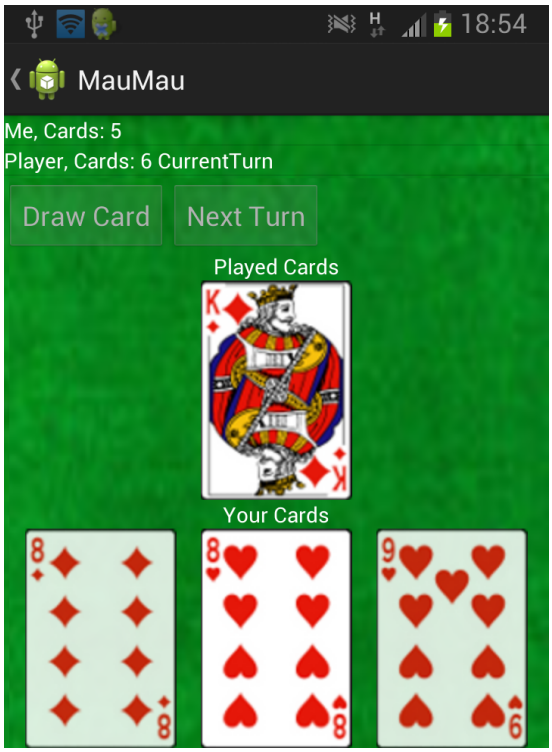


Abbildung 5.1: MauMau

Die Regeln in dieser implementierten Version sind folgende:

- Man darf entweder die Spielfarbe oder die/den Zahl/Wert spielen, von der Karte, welche zuvor gespielt worden ist. So dürfte man eine beliebige Neun oder eine beliebige Karo Karte spielen, wenn davor eine eine Karo Neun gespielt wurde.

- Falls davor eine Acht gespielt wurde, muss man einen Zug aussetzen und der nächste ist dran, oder man legt selber eine Acht, falls man eine hat und somit würde der Nächste aussetzen.

- Falls davor ein Ass gespielt wurde, muss man einen Zug aussetzen und der nächste ist dran, oder man legt selber einen Ass, falls man einen hat und somit würde der Nächste aussetzen. Also das gleiche wie bei Acht.

- Spielt ein Spieler eine Sieben, so muss der nächste Spieler 2 Karten ziehen, oder er hat eine Sieben und spielt sie anstatt. Falls anstatt des Ziehens eine Sieben gelegt wurde muss der nächste Spieler dann schon 4 Karten ziehen, bis keiner mehr eine Sieben spielen kann.

- Falls man einen Buben spielt, darf eine Spielfarbe wählen, welche der nächste Spieler dann spielen muss.

- Wenn man eine Karte nicht spielen kann, z.B. falls man keine passende Karte hat, muss man eine Karte ziehen. Wenn man dann immer noch keine passende Karte hat, setzt man den Zug aus und der Nächste ist dran.

- (Optional) man sagt laut MauMau, bevor man seine letzte Karte spielt.

5.2 Anforderungen

Eine Liste mit den Anforderungen, welche zur Implementierung des Spiels notwendig sind, sieht folgendermaßen aus:

- Man sieht alle Spieler, die zum Spiel verbunden sind.
- Man sieht wie viel Karten jeder dieser Spieler besitzt.
- Jeder Mitspieler sieht die zuletzt gespielte Karte.
- Man kann nur die Karten spielen, welche vom Regelwerk her erlaubt sind.
- Man spielt eine Karte, indem man das Bild der Karte berührt.
- Wenn man einen Buben spielt, erscheint ein Dialog, bei den man die Spielfarbe wählen kann.
- Es gibt Buttons um Karten zu ziehen und um einen Zug auszusetzen, welche der Situation entsprechend aktiviert/deaktiviert sind.
- Eine Nachricht erscheint, falls einer der Spieler gewonnen hat.

5.3 Implementierung

5.3.1 Projektkonfiguration

Die Projektkonfiguration unterscheiden sich bis auf die Namensgebungenen kaum von der im Kapitel 4.3.1. Somit wird hier nicht mehr darauf eingegangen.

5.3.2 MauMauApplication

Die ***MauMauApplication*** Klasse erbt von ***Application*** und ist somit die Klasse, welche am Anfang gestartet wird. In der ***onCreate()*** Methode wird der ***GameManager*** instanziiert und der ***PTPHelper*** initialisiert. Wie der ***PTPHelper*** genau initialisiert wird sieht man im Codeausschnitt 5.1.

```
1 PTPHelper.initHelper("MauMau",GameManagerInterface.class, this, new
      GameManagerDummyObject(), gameManager, MauMauLobbyView.class);
```

Codeausschnitt 5.1: MauMau PTPHelper Initialisierung

Es werden der Applicationsname, die Inteface Klasse und der ***Context***, welches als ***this*** angegeben ist, als ersten übergeben. Darauf folgt das ***BusObject***, welches nur eine leere Implementierung des Intefaces ist. Schließlich wird noch der ***gameManager***, welches den ***SignalHandler*** representiert, und die ***MauMauLobbyView*** Klasse übergeben. Der letzte Parameter ist die konkrete Implementierung der ***LobbyActivity***, die nur angibt, welche Views geöffnet werden sollen. Die ***MauMauApplication*** implementiert auch den ***GameManagerObserver***, welcher die Methode ***update(int)*** definiert. Diese Methode wird von den ***GameManager*** aufgerufen um der ***MauMauApplication*** mitzuteilen, verschiedene Nachrichten an die Mitspieler zu senden. Über das Argument wird der Nachrichtentyp mitgeteilt und an den ***BackgroundHandler*** weitergeleitet, der über einen eigenen Thread die Informationen vom ***GameManager*** erfragt und weiterschickt. Ein Beispiel der Methode ***sendPlayedCard()*** um die Funktionsweise verständlich zu machen, sieht man im Codeausschnitt 5.2.

```
1  private void sendCardPlayed() {
2   Card playedCard = gameManager.getPlayedCard();
3   GameManagerInterface gameManagers = (GameManagerInterface) PTPHelper.
        getInstance().getSignalEmitter();
4   if(gameManagers!=null){
5     try {
6       gameManagers.PlayCard(playedCard.id, PTPHelper.getInstance().getUniqueID())
            ;
7     } catch (BusException e) {
8       e.printStackTrace();
9     }
10  }
11 }
```

Codeausschnitt 5.2: sendCardPlayed Methode des BackgroundHandlers

Der ***BackgroundHandler*** führt die Methode aus und holt sich in diesem Fall vom ***GameManager*** die gespielte Karte. Daraufhin wird der ***SignalEmmiter*** vom ***PTPHelper*** geholt und darüber wird die gespielte Karten Anderen mitgeteilt. Weite Methoden des BackgroundHandlers sind ***notifyPeersAboutMe(), sendCardOwnerChanged(), sendOwnedCards()*** und ***sendNextTurn()***.

5.3.3 GameManager

Der ***GameManager*** ist das Herzstück des Spiels und beinhaltet das Datenmodel und alle Informationen, welche für das Spiel wichtig sind. Nachdem der GameManager instantiiert wurde, wird die ***reset()*** Methode aufgerufen. Die ***reset()*** Methode setzt alle Werte auf den Anfangswert und initialisiert alle Karten mit Werten und den entsprechenden Bitmaps. Die Bitmaps der Karten werden aus einem Bild entnommen auf dem alle Karten sortiert abgebildet sind, indem durch den Wert ein Offset ermittelt wird und daraus die Position des Bildes dieser Karte. Der Codeausschnitt 5.3 zeigt wie die Bitmap genau erstellt werden.

```
1  public Bitmap getBitmap(Context context,Card card){
2    if(allCards == null){
3      allCards = BitmapFactory.decodeResource(context.getResources(),
4              R.drawable.cards);
5    }
6    int cardWidth = allCards.getWidth()/13;
7    int cardHeight = allCards.getHeight()/5;
8    int value = card.value;
9    if(value == 14) value = 1; //Ace is positioned at the beginning of the bitmap
10   Bitmap cardBitmap = Bitmap.createBitmap(allCards,(value-1)*(cardWidth), (card.
         suit)*(cardHeight),cardWidth, cardHeight);
11   return cardBitmap;
12 }
```

Codeausschnitt 5.3: Bitmaps den Karten zuweisen

Der ***GameManager*** besitzt einen sogenannten ***RuleEnforcer***, welcher dafür zuständig ist Situationsbedingt verschiedene Regeln anzuwenden. Der ***RuleEnforcer*** besitzt zwei Listen von ***Rules***, wobei jedes dieser ***Rules*** eine Methode ***boolean isAllowed(Card)*** implementiert. Eine Liste beinhaltet inklusive Regeln, also Regeln, die alle erfühlt sein müssen. Die zweite Liste beinhaltet die exklusiven Regeln, bei denen mind. eine Regeln erfühlt sein muss. Ein Beispiel dafür ist die Regel, dass man am Zug ist. Diese Regel muss immer gelten, wenn man eine Karte spielen will, so geht diese Regel in die inklusive Liste. Die Regeln, dass man eine Spielfarbe oder einen bestimmten Wert bedienen muss, gehören in die exklusive Liste, da nur eine dieser Regeln erfühlt sein muss um eine Karte spielen zu können. Diese Liste wird z.B. am Ende jedes Zuges aktuallisiert, wie im Codeausschnitt 5.4 aus der Methode ***NextTurn*** zu sehen ist.

```
1 CardPlayedEvent cardPlayedEvent = new CardPlayedEvent(playedCard,
      playCardRuleEnforcer,this);
2 cardPlayedEvent.updateRuleEnforcer();
```

Codeausschnitt 5.4: Aktuallisierung der Regeln

Der ***GameManager*** ist ein ***SignalHandler*** und implementiert somit einige Methoden um mit den anderen Mitspielern zu kommunizieren. Die Auflistung der Methoden sieht man im Codeausschnitt 5.5.

```
1  @BusSignal
2  public void ChangeOwner(int cardId, String uniqueUserID) throws BusException;
3  @BusSignal
4  public void PlayCard(int cardId, String uniqueUserID) throws BusException;
5  @BusSignal
6  public void NextTurn(String uniqueUserID,int specialCase) throws BusException;
7  @BusSignal
8  public void HiIAm(String uniqueID,String playerName) throws BusException;
9  @BusSignal
10 public void ByeIWas(String uniqueID,String playerName) throws BusException;
```

Codeausschnitt 5.5: Methoden des GameManagerInteface

Die Methode ***HiIAm*** wird von jeden Mitspieler aufgerufen, der sich zu einem Spiel verbindet, sodass alle diesen Spieler kennen und in die Mitspielerliste einfügen können. Die Methode ***ByIWas*** hingegen wird aufgerufen um mitzuteilen, wenn ein Mitspieler die Runde verlässt und somit aus der Mitspielerliste entfernt werden kann. ***NextTurn*** wird von dem Spieler aufgerufen, der gerade dran ist und den Spielzug beendet hat. Ein Parameter ist immer die ***uniqueID***, welche den Sender repräsentiert, damit man weiß von wem die Nachricht stamm. Die ***NextTurn*** Methode hat noch den zweiten Parameter nämlich den ***specialCase***, welcher bestimmte Situationen kennzeichnet. Es wird z.B. nur die letzte Karte gespeichert, die gespielt wurde. Somit kann man nicht mehr nachvollziehen ob 2 oder 3 Siebenen aufeinander gespielt wurden und der Spieler z.B. 6 Karten anstatt nur 2 ziehen soll. Diese Situation wird über diesen Parameter übergeben, also wenn ***specialCase=3*** heißt es, dass die Sieben 3 mal aufeinander gespielt wurde. Dann gibt es noch die ***ChangeOwner*** Methode, die dafür zuständig ist den Kartenstatus synchron zu halten. D.h. wenn jemand eine Karte zieht muss jeder GameManager davon erfahren, sodass zwei Spieler nicht die selbe Karten ziehen können. Schließlich gibt es noch die ***PlayCard*** Methode, welche allen Mitspielern mitteilt, welche Karte gespielt wurde und angezeigt werden kann.

5.3.4 GameActivity

Die ***GameActivity*** ist zum Einen für die UI zuständig und zum Anderen für die graphische Darstellung des Spiels. Die ***onCreate()*** Method beinhaltet die ganzen Initialisierung der Buttons und andere UI Elementen wie z.B. der ***Gallery***. Die ***Gallery*** ist aus dem Android SDK und ist dafür gedacht mehrere Bilder darzustellen und stellt vorallem Funktionen bereit wie das Scrollen und Auswählen von diesen. Um die ***Gallery*** mit Bildern zu füllen benötigt man einen ***BaseAdapter***, was im Prinzip das Model darstellt. Somit braucht man nur das Model updaten und die ***Gallery*** zeigt den aktuellen Stand, z.B. alle Karten, die man gerade besitzt. Dann gibt es noch den ***OnItemClickListener***, welcher für die ***Gallery*** implementiert wird. Dieser Listener implementiert die Aktionen, welche ausgeführt werden sollen, falls man eine Karte berührt. Ein Auschnitt aus der ***OnItemClick*** Methode sieht man im Codeausschnitt 5.6:

```
1 if(gameManager.canPlayCard(card)){
2   if(card.value == 11){
3   cardToPlay = card;
4   showWishSuitWindow(card);
5   }else{
6   playCard(card);
7   }
8 }
```

Codeausschnitt 5.6: Karte spielen

Hierbei wird zuerst der ***gameManager*** gefragt ob die Karte überhaupt gespielt werden kann, indem es mit dem ***RuleEnforcer*** geprüft wird. Danach wird noch geprüft ob sich bei der Karte um einen Buben handelt und ein Fenster angezeigt werden soll um sich eine Spielfarbe zu wünschen, was man z.B. auf dem Bild 5.2 sehen kann. Ansonsten wird die Karte einfach gespielt, in dem die Karten-ID dem ***GameManager*** mitgeteilt wird. Die ***GameActivity*** implementiert auch den ***GameManagerObserver*** Inteface, sodass es auch eine ***update(int)*** Methode gibt. Dadurch kriegt die ***GameActivity*** z.B. mit ob eine neue Karte gespielt wurde oder sich ein neuer Spieler verbunden hat und die UI neu gerendert werden muss. Diese Nachrichten können jedoch auch

Abbildung 5.2: Spielfarbe wünschen

vom BusThread kommen, wenn z.B. eine Nachricht im ***GameManager*** von anderen Mitspielern eintrifft. Daher müssen alle UI bedingten Operationen an den UI Thread weitergeleitet werden. Wie das gemacht wird sieht man im Codeausschnitt 5.7

```
case GameManager.NEXT_TURN: runOnUiThread(new Runnable() {
  @Override
  public void run() {
    playerListAdapter.refresh();
    playerListAdapter.notifyDataSetChanged();
    updateButtonsState();
  }
});
```

Codeausschnitt 5.7: Operationen auf dem UI Thread ausführen

Durch die Methode ***runOnUiThread*** kann man beliebige Operationen auf dem UI Thread ausführen. Dieser Fall behandelt das Ende eines Zuges. Hier wird zum einen die Spielerliste aktuallisiert, da sie anzeigt wer gerade am Zug ist, und die Buttons werden entsprechend auf ***enabled/disabled*** gesetzt, abhängig ob man gerade am Zug ist oder nicht. Es gibt bei Android auch die Möglichkeit auf den Zurückbutton zu drücken,was dafür sorgt,dass man in die vorherige View gelangt. Dies muss man jedoch dem ***GameManager*** mitgeteilt werden, sowie dem ***PTPService***, da man dadurch das laufende Spiel verlässt. Der Codeausschnitt 5.8 zeigt die Behandlung dieses Zurückbuttons.

```
@Override
public void onBackPressed(){
  super.onBackPressed();
  gameManager.reset();
  PTPHelper.getInstance().leaveChannel();
  PTPHelper.getInstance().disconnect();
  PTPHelper.getInstance().connectAndStartDiscover();
}
```

Codeausschnitt 5.8: Behandlung des Zurückbutton

Dabei wird die ***onBackPressed()*** Methode von der ***Activity*** überschrieben. Weiterhin wird der ***gameManager*** zurückgesetzt, sodass man alles wieder mit Anfangswerten initialisiert und man ein neues Spiel starten kann. Zum Schluss wird noch dem ***PTPHelper*** mitgeteilt die laufende Session zu schliessen und eine neue Verbindung aufzubauen, sodass man im Anfangszustand sich befindet, da man durch die Zurücktaste in der Lobby landet.

6 Fazit

Die Entwicklung von Peer-to-Peer Spielen im Lokalen Wlan ist eine herausfordende und spannende Sache. Mehrspieler Spiele ohne einen Server sind heutzutage noch nicht sehr weit verbreitet und das bietet daher sehr viele Möglichkeiten neue Konzepte und Ideen zu realisieren. Die Herausforderung liegt in der Komplexität der Implementierung dieser Spiele, sowie in den Beschränkungen, welche durch den größeren Synchronizationsaufwand entstehen. Besonder an dem Praktischen Beispiel MauMau sah man, dass es viele Fälle gibt, die man beachten muss um den Zustand synchron zu halten und daher war es notwendig mehr Informationen über das Netzwerk zu verschicken, was zu mehr Datenverkehr führt. Das Rahmenwerk AllJoyn hielt was es versprach, mit all seinen Funktionalitäten, bis auf den leichten Einstieg, denn gerade das Verstandniss des Mechanismus ist anfangs recht mühsam und bedarf viel Ziel. Eine weitere Problematik kommt aus der Tatsache, dass der Android Emulator, welcher mit dem Android SDK mitgelieft wird, keine Wlan Unterstützung besitzt und somit die Entwicklung mithilfe des Emulator nicht möglich ist. Dies führt dazu, dass für die Entwicklung von Mehrspielerspielen mindestens 2 Android Geräte notwendig sind um das Spiel vernünftig testen zu können. Daher war es auch schwer Belastungstest durchzuführen, da dafür eine entsprechende Anzahl an Geräten benötigt wird. Alternativ gäbe es noch die Möglichkeit eine Android x86 Version[4] auf einer VM laufen zu lassen, jedoch ist die native Bibliothek von AllJoyn nur für die ARM-Architektur verfügbar. Doch auch hier gibt es eine Möglichkeit AllJoyn für die x86 Architektur zu bauen, da der Quellcode frei verfügbar ist, was jedoch einen großen Aufwand bedeutet und die Auseinandersetzung mit dem Android Quellcode. Trotz der Herausforderungen und Probleme mit dem Peer-to-Peer Prinzip, besteht das Interesse für eine unabhängige Kommunikation zwischen Mobilen Endgeräten. Dies zeigt z.B. Samsung mit Samsung Chrod SDK[5], welches im Prinzip dem AllJoyn SDK ähnlich ist, bis auf die Tatsache, dass es nicht Opensource ist und unter Samsungs License steht, was die Entwicklung etwas einschränken kann. Aber das Interesse von Samsung bestätigt, dass ein gewisser Potential in der Entwicklung von Peer-to-Peer Anwendung und Spielen besteht.

A Literaturverzeichnis

[1] Android SDK http://developer.android.com [Stand: 11.07.2013]

[2] AllJoyn Framework http://www.alljoyn.org [Stand: 11.07.2013]

[3] Peer-to-Peer http://www.itwissen.info/definition/lexikon/Peer-to-Peer-Netz-P2P-peer-to-peer-network.html [Stand: 11.07.2013]

[4] Porting Android to x86 http://www.android-x86.org/documents/virtualboxhowto [Stand: 09.07.2013]

[5] Samsung Chord http://developer.samsung.com/chord [Stand: 09.07.2013]

[6] Managing Android Projects http://developer.android.com/tools/projects/index.html [Stand: 03.07.2013]

[7] Algorihmus zur Berechnung von Linienüberschneidungen http://www.java-gaming.org/index.php?topic=22590.0 [Stand: 03.07.2013]

[8] 900 Millionen Android Geräte aktiviert http://www.androidinside.de/android-900-millionen-geraete-aktivierungen-und-48-milliarden-app-installationen/ [Stand: 18.07.2013]

[9] Abbildung zur Veranschaulichung des AllJoyn Bussystems https://www.alljoyn.org/app-developers/getting-started [Stand: 18.07.2013]