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

Lokales Wlan basiertes Multiplayer Framework für Android

BACHELORARBEIT

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, ohne die Verwendung eines externen Servers. Für die Verbindung zwischen den Smartphones ist ein bereits vorhandenes Rahmenwerk namens AllJoyn[2] verwendet worden. AllJoyn ermöglicht eine Kommunikation der Smartphones über das Funknetzwerk. Die Bibliothek ist für die Entwicklung von Spielen für Android bestimmt und basiert daher dem Android SDK[1]. Durch die Verwendung der Bibliothek wird die Kommunikation zwischen mehreren Smartphones soweit vereinfacht, dass bei der Implementierung nur auf die Spielmechanik konzentriert werden kann. Funktionen wie die Vermittlung, das Erstellen und das Verbinden zum Spiel, wird von der Bibliothek übernommen. Im Anschluss ist das Kartenspiel MauMau und ein Puzzelspiel GraphenGame implementiert worden. Die beiden Spiele dienen als Demonstration der Funktionalität dieser Bibliothek.

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 **3**

2 Grundlagen **4**

2.1 Konzepte 4
2.2 Peer-to-Peer 4
2.3 Android SDK 5
2.3.1 Activity 6
2.3.2 Service 6
2.3.3 Application 7
2.3.4 Context 7
2.3.5 View 7
2.3.6 Handler 7
2.3.7 UI-Thread 7
2.3.8 AndroidManifest 8
2.4 AllJoyn 9
2.4.1 Bussystem 9
2.4.2 BusAttachement 10
2.4.3 Interface 11
2.4.4 ProxyObjecte 11
2.4.5 BusObject 11
2.4.6 SignalHandler 12
2.4.7 SignalEmitter 12
2.4.8 Session 13

3 Implementierung **14**

3.1 Realisierung 14
3.2 PTPHelper 14
3.3 LobbyActivity 20
3.4 PTPService 20
3.4.1 doConnect() und doDisconnect() 21

3.4.2 doStartDiscovery() und doStopDiscovery() 23
3.4.3 doBindSession() und doUnbindSession() 23
3.4.4 doRequestName() und doReleaseName() 25
3.4.5 doAdvertise() und doCancelAdvertise() 26
3.4.6 doJoinSession() und doLeaveSession() 26
3.4.7 doQuit() 27

4 Praktisches Anwendungsbeispiel: Graphenspiel 28
4.1 Spielidee 28
4.2 Implementierung 29
4.2.1 Projektkonfiguration 29
4.2.2 MainApplication 29
4.2.3 Graph 31
4.2.4 DrawView 32

5 Praktisches Anwendungsbeispiel: MauMau 34
5.1 Spielidee 34
5.2 Anforderungen 36
5.3 Implementierung 36
5.3.1 Projektkonfiguration 36
5.3.2 MauMauApplication 36
5.3.3 GameManager 38
5.3.4 GameActivity 40

6 Fazit 42

A Literaturverzeichnis I

1 Einleitung

Spiele für Smartphones gibt es unzählige. Die meisten dieser Spiele sind entweder Solospiele, oder sind für mehrere Spieler unter Verwendung eines Servers. Trotz der mittlerweile gut ausgebauten Netzabdeckung und guter Internetgeschwindigkeit auf Smartphones, kann das Spielerlebniss durch eine schwankende Internetgeschwindigkeit getrübt werden. Dies ist von der Unterschiedlichen Netzabdeckung der einzelnen Mobilfunkanbieter abhängig.[11] Actionspiele, die eine schnelle Antwortzeit benötigen, setzen somit mindestens eine ***HSDPA***[10] Verbindung vorraus. Einige Spiele senden außerdem eine große Menge an Daten an den Server. Das kann, abhängig vom vorhandenen Vertrag, das Datenvolumen schnell verbrauchen. Es gibt auch Situtationen, bei denen es keine oder nur eine beschränkte Internetverbingung gibt, z.B. im Ausland oder bei einer Zugfahrt. Für diese Situtationen gibt es die Möglichkeit eine lokale ***Peer-to-Peer***[3] Verbindung zwischen den Smartphones zu erstellen. Das Interesse eine lokale Verbingung zwischen Smartphones herstellen zu können, wird auch an den Bemühungen seitens Samsung erkennbar. Samsung hat 2013, während der Erstellung dieser Arbeit, das ***Chord SDK***[5] herausgebracht, das die Kommunikation zwischen einzelnen Samsung Smartphones über das lokale Funknetz ermöglicht. Das Chord SDK steht jedoch unter der Samsung License und ermöglicht nur die Entwicklung auf Samsung Smartphones. Die Entwicklung einer Bibliothek, die ***OpenSource*** ist und die Verwendung auf allen Android Geräten erlaubt, bietet somit eine gute Alternative. Durch diese Bibliothek wird die Entwicklung der Spiele soweit vereinfacht, dass bei der Entwicklung nur auf die Spiellogik geachtet werden muss. Die Verbindungtechnischen Prozesse werden von der Bibliothek übernommen. Für die Entwicklung eines Spiele auf Basis der Bibliothek wird nur ein Android Gerät benötigt. Die Bibliothek kann auch für praktische Anwendungen, wie z.B. Textmessanger, verwendet werden. Es wird jedoch im Rahmen dieser Ausarbeitung nur auf die Realisierung von Spielen eingegangen. Die Bibliothek wird im weiteren Verlauf der Arbeit als ***PTPLibrary*** bezeichnet.

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 Smartphones hat eine identische Version des Spiels, sodass die Organisation im Spiel realisiert wird. Wie in der Abbildung 2.1 zu sehen, kann jedes Spiel auf einem Smartphone mit den anderen Smartphone 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 Spieler kann eine Session erstellen, zu der sich die anderen Spieler verbinden, um an der Spielrunde teilnehmen zu können. Durch die Session wird die Möglichkeit gegeben mehrere Spiele simultan im selben Netzwerk zu spielen.

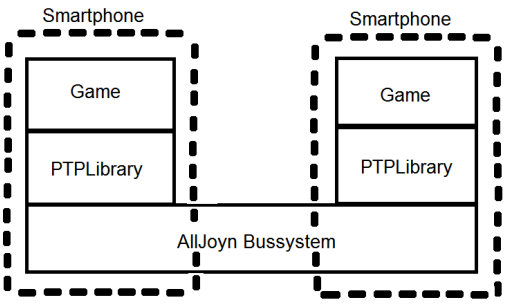
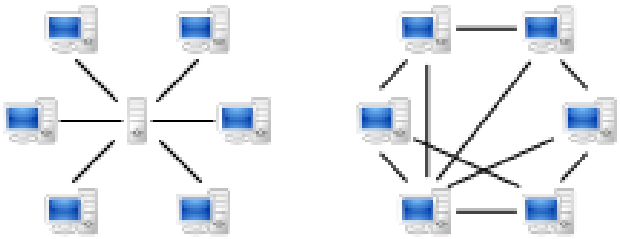


Abbildung 2.1: Architektur-Konzept

2.2 Peer-to-Peer

Peer-to-Peer kommt vom englischen Wort peer=Gleichberechtigter und entspricht somit einem Netzwerk aus gleichberechtigten Teilnehmern. Durch das Weglassen eines Servers, ist die Organisation den Peers überlassen.



(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*** zeigen die Unteschieden zwischen einem Serverbasierten und einem Peer-to-Peer Netzwerk. Durch das Weglassen eines fest zugeordneten Servers, wird eine höhere Flexibilität erreicht, sodass nur Peers benötigt werden um das Spiel zu spielen. Durch ein Peer-to-Peer Netzwerk entstehen jedoch auch Probleme und Herausforderungen. 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. Rechenintensive Aufgaben können vom Server übernommen werden und so die einzelnen Smartphones entlasten. Bei einem Peer-to-Peer System hingegen muss der Zustand von jedem einzelnen Smartphone ermittelt und mit den anderen Smartphones synchonisiert werden. Dies hat einen höheren Ressourcenaufwand und kann die Anzahl der möglichen Teilnehmer limitieren. Jedoch ist es auch vom Anwendungfall abhängig, denn ein entsprechendes Gegenbeispiel für eine hohe Peersanzahl zeigten Tauschbörsen wie Torrent oder Napster, um nur zwei bekannte Beispiele zu nennen. Die Entwicklern müssen zum einen entscheiden 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, das 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 es wird klar, 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, das auch als das Android Developer Tool kurz ADT verfügbar ist. ADT ist ein Plugin für das mittlerweile weit verbreitete Entwicklungsumgebung Eclipse, welche die Entwicklung und die Übertragung der Applikation auf das Gerät problemlos ermöglicht. Weiterhin bringt das Android SDK einen Emulator mit sich, das 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. 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, die die Erstellung von einzelnen UI-Fenstern übernimmt. Die meisten Apps in Android bestehen 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, die dazu gedacht ist Hintergrundprozesse zu übernehmen. Ein Service wird von z.B. einer Activity gestarted und läuft dabei im Hintergrund, sogar wenn die Activity nicht mehr existiert. Ein Service ist gut dafür geeignet um z.B. die Netzwerkkommunikation im Hintergrund zu behandeln, ohne die Applikation selbst zu behindern. Durch das Binden des Services an eine Applikation, wird der Service beended, wenn die Applikation beendet wird.

2.3.3 Application

Application bietet zusätzlich zu den Activities die Möglichkeit während der ganzen Laufzeit der Applikation eine feste Instanz zu haben, die den Zustand bestimmter Daten beinhaltet. Es kann mit einem Singleton verglichen werden, der den Status der Applikation beinhaltet. Durch die Methode ***Context.getApplicationContext()***, kann auf die Application Instanz zugegriffen werden.

2.3.4 Context

Der Context beinhaltet Informationen über die Applikationsumgebung und lässt verschiedene Aktionen zu. Eines dieser Aktionen ist 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, die das UI-Fenster representiert oder können die UI-Elemente direkt über das XML-Layout laden.

2.3.6 Handler

Ein ***Handler*** ist ein Object, das 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 gut für die Kommunikation zwischen verschiedenen Thread geeignet.

2.3.7 UI-Thread

Der UI Thread ist der Hauptthread, der 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.

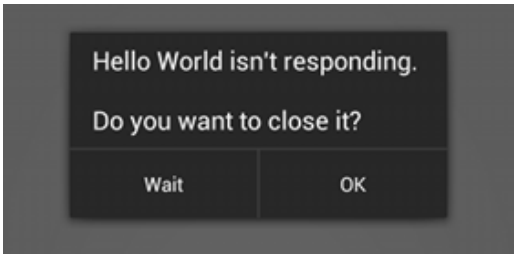


Abbildung 2.3: ANR

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 der Verwendung von mehreren Thread darauf geachtet werden, dass die UI Komponenten nur vom UI Thread angefasst werden. Es gibt die Möglichkeit bei Background Threads die ***runOnUiThread()*** Methode zu verwenden, die die auszuführende Aufgabe an den UI Thread übergibt.

2.3.8 AndroidManifest

Das Android Manifest ist eine XML-Datei, die 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, das 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 Frameworks besteht in der Tatsache, dass es verschiedene Betriebssysteme unterstützt und eine Vielzahl von Programmiersprachen, darunter C♯, C++, Java sowie Objective C. Mit AllJoyn können Anwendungen auf Peer-to-Peer Basis für Windows,MacOS,Linux,Android und iOS entwickelt und miteinander verbunden werden. Es bietet weiterhin die Unterstützung für Bluetooth, Wifi und Ethernet. AllJoyn ermöglicht außerdem das Finden und die Verbindung von Mobilen-Endgeräten im Wifi-Netz. Das AllJoyn Rahmenwerk übernimmt somit alle Aufgaben auf der Netzwerkschicht. Zusätzlich bietet AllJoyn für jedes Betriebssystem ein SDK an, das alle notwendigen Bibliotheken und 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 verdeutlicht. Jedes dieser Anwendung muss einen ***BusAttachement*** erstellen und den entsprechenden EventHandler implementieren, über den die Nachrichten von anderen Geräten behandelt werden. Es können Methoden direct auf einem entfernten Object ausgeführt werden. Dieses Objekt wird in der Abbildung als ***ProxyObject*** dargestellt. Es könenn auch Nachrichten an alle Teilnehmer gleichzeitig geschickt werden, die von einen ***EventHandler*** behandelt werden. Der ***EventHandler*** muss das entsprechende Interface implementieren, das die Mehoden definiert, über die kommuniziert wird. Das genauere Prinzip des ***BusInterface*** wird im weiteren Verlauf genauer erläutert. Der ***Bus*** selbst ist ein sogenannter ***Daemon***,

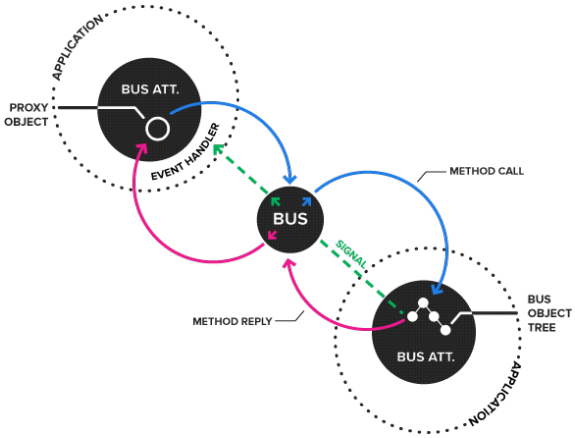


Abbildung 2.4: Bussystem [9]

also ein Hintergrundprozess, der auf dem Gerät läuft und sich um netzwerktechnischen Aufgaben kümmert. Dieser wird z.B. über einen Aufruf in der Anwendung gestartet und bietet durch den ***BusAttachement*** eine Schnittstelle für Anwendungen um damit zu interagieren.

2.4.2 BusAttachement

Um mit dem Bussystem zu kommunizieren, wird ein ***BusAttachement*** benötigt. Dieser muss erstellt werden, 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. AllJoyn ermöglicht es mehrere Anwendungen auf dem selben Gerät laufen zu lassen, sodass sie untereinander kommunizieren können. Der zweite Parameter gibt an, ob der 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, die zu den selben Bus verbunden sind, der 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.

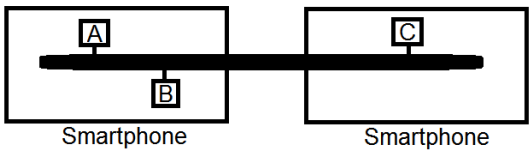


Abbildung 2.5: Bus

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, die 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

Ein Proxy Objekt ist ein Vertreter eines anderen Objektes, das z.B. auf einem anderen Gerät liegt. Mithilfe von Proxy Objekten können Methoden auf diesen entfernten Objekten ausgeführt werden. Sie implementieren das zuvor definierte Interface, worüber dann die Kommunikation realisiert wird. Durch die Angabe eines Intefaces, kann der BusAttachment das entsprechende ProxyObjekt zurückliefern.

2.4.5 BusObject

Ein BusObject ist ein Objekt, das ein definierte Interface implementiert und unter einem bestimmten Pfad abgespeichert wird. Der Pfad ist eine busweit einzigartige Zeichenkette, die in Form eines Dateipfades dargestellt wird. Über diesen Pfad kann mithilfe des BusAttachements das entsprechende ProxyObject erfragt werden.

2.4.6 SignalHandler

Ein SignalHandler ist ein Object, das die Behandlung von eingehenden Nachrichten implementiert. Der Unterschied zwischen einem BusObject und einem SignalHandler ist, dass beim BusObject nur die Methode auf dem entsprechenden Object ausgeführt wird, der 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 bei der Implementierung eine weitere Annotation hinzugefügt werden 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. Die ***id*** ist die SessionID, die beim Verbinden zu einer Session mitgeteilt wird. Zusätzlich lässt sich noch konfigurieren, ob das Signal auch über den Bus hinaus weitergeleitet wird, falls mehrere Bussysteme gleichzeitig laufen. 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 behandeln. Dazu muss ein ***Channel*** erstellt werden, zu dem sich alle anderen Teilnehmen verbinden können. Dafür benötigen alle Geräte zum einen den Namen des Channels und zum anderen die ***Channelportnummer***. Über das ***Advertising*** kann der Host den erstellten Channelname den anderen Teilnehmern mitteilen. Um diese Mitteilungen zu empfangen, müssen die Teilnehmer einen ***BusListener*** implementieren, der auf diese Nachrichten reagiert. Das Erstellen der Session benötigt unter einigen Einstellungen, wie die Portnummer,Transportprotokol usw, auch den ***SessionPortListener***, der z.B. das Verbinden von anderen Teilnehmern behandelt. Zwei wichtige 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

3 Implementierung

3.1 Realisierung

Die PTPLibrary bietet eine einfache Schnittstelle, über die eine Kommunikation mit allen Teilnehmern ermöglicht wird. Es ist außerdem eine Activity implimentiert worden, welche das Erstellen und das Verbinden zu Spielen übernimmt. Diese Activity besitzt eine graphische Oberfläche und kann bei der Implementierung eines Spiels verwendet werden. Diese Activity wird als ***Lobby*** bezeichnet. Die Netzwerkverbindung von AllJoyn läuft in einem seperaten Thread und ist daher mithilfe von ***Handlern*** von dem ***UI-Thread*** getrennt behandelt worden, sodass sich die beiden Threads nicht behindern. Für die Entwicklung der PTPLibrary und der Beispielanwendungen sind 2 Android Smartphones verwendet worden. Das erste Gerät ist ein ***Samsung Galexy S3 GT-I9300*** mit dem Betriebssystem Android 4.1.2. Das zweite Gerät ist ein ***Sony Xperia Tipo ST21i***, das das Betriebssystem Android 4.0.2 besitzt. Die Verbindung ist mithilfe eines ***WiFi AccessPoint*** von Android erstellt worden, zu dem sich das andere Gerät verbunden hat. Um die Bibliothek benutzen zu können muss der Entwickler nur die PTPLibrary als Bibliothek in sein Projekt einbinden.

3.2 PTPHelper

Der PTPHelper ist eine Schnittstelle zwischen dem Spiel und dem Hintergrundservice, der sich um die Verbindung kümmert. Nach Abwägen mehrerer Möglichkeiten wie der Service, der sich um die Verbindungstechnischen Abläufe kümmert, in die Applikation integriert werden kann, ist die Wahl schließlich auf die Benutzung eines Singletons gefallen. Alternativ ließe sich eine abstrakte Application benutzen, die der Entwickler in seinem Projekt implementieren müsste, aber dies würde dem Entwickler auch eine Architekturentscheidung aufzwingen. Der Entwickler kann entscheiden, wann der Service gestartet wird und wie damit interagiert werden soll. So

kann z.B. der Einzelspielermodus vollkommen ohne der Benutzung des PTPHelpers realisiert werden und erst beim Multiplayermodus wird der Service gestartet. Um den PTPHelper zu initialisieren muss der Entwickler in seinem Code den folgenden Befehl aus dem Codeausschnitt 3.1 ausführen.

```
1 PTPHelper.initHelper("appName", context,MyLobby.class);
```

Codeausschnitt 3.1: Helper initialisieren

Der Parameter ***appName*** ist der Name der Application und legt den Namen für das ***BusAttachement*** fest. Beim ***context*** handelt es sich um die Android Context Klasse, für das Starten des Services. Der ***MyLobby.class*** Parameter gibt an, welche Activity als Lobby gestartet werden soll. Daraufhin kann über den Getter die initialisierte Instanz gefragt werden, wie im Codeausschnitt 3.2 gezeigt.

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

Codeausschnitt 3.2: Getter für die initialisierte Instanz

Es muss jedoch beachten werden, das das Starten des Hintergrundservices ein asynchroner Prozess ist, und der Hintergrundservice nach der Initialisierung nicht vollständig gestartet gestartet ist. Um den Zustand des Hintergrundservices zu erfahren kann über den Methodenaufruf, wie im Codeausschnitt 3.3 gezeigt,dieser erfragt werden.

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

Codeausschnitt 3.3: Verbindungstatus erfragen

Dabei handelt es sich um einen Integer, der die Werte CONNECTED=7 und DISCONNECTED=8 annehmen kann. Damit der Service nur so lange läuft wie die Application wird der Service 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 um einen Listener, der 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 Smartphone über eine WLAN 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 WLAN-Verbindung gibt, wird die Initialisierung abgebrochen und der Service wird nicht gestartet. Android ermöglicht es zusätzlich ein ***AccessPoint*** zu erstellen, zu den sich andere Smartphones verbinden können. Die Information über den Zustand des AccesPoints ist jedoch keine öffentliche Methode des WiFi-Manangers, sodass diese Information nur über Java-Reflection erfragt werden kann. 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

Nachdem der PTPHelper initialisiert ist und der Service gestartet, kann über den PTPHelper den Service mitgeteilt werden z.B. sich zu einer Session zu verbinden oder eine Session zu erstellen. Dies geschieht über Methodenaufrufe, wie im Beispiel 3.7 gezeigt.

```
1 //Host
2 PTPHelper.getInstance().setHostSessionName(name)
3 PTPHelper.getInstance().hostStartSession()
4 ...
5 //Client
6 PTPHelper.getInstance().setClientSessionName("sessionName");
7 PTPHelper.getInstance().joinSession();
```

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, die der Entwickler selber implementieren kann. Zu den LobbyObservern gehört z.B. auch die AbstractLobbyActivity und implementiert die Methode ***conncetionStateChanged(int state)***, die 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 sessionName);
2 public void lostAdvertisedName(String sessionName);
3 public void busDisconnected();
```

Codeausschnitt 3.8: BusObserver

Durch die Implementierung des BusObservers kann auf das Öffnen und Schließen der Spiele reagiert werden, 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 ***foundSessions*** gespeichert und entsprechend auch entfernt werden. 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. Dies ermöglicht es dem Entwickler bestimmte Aktionen auszuführen, falls neue Spieler sich zur einer Session verbinden, sich von der Session trennen oder die Verbindung zur Session abbricht. Schließlich gibt es noch den ***ServiceHelperObserver***, der z.B.vom PTPService implementiert wird. Dieser Observer hört auf den Helper, der über die Methode ***doAction(int action)*** verschieden

Aktionen mitteilt. So hat z.B. der Methodenaufruf aus Codeausschnitt 3.23

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

Codeausschnitt 3.9: verbinde zur Session

zur Folge, dass der PTPHelper den Service über den Observer mitteilt sich zu einer Session zu verbinden. Die oben genannten Observer sind mithilfe eines ***Handlers*** vom Hintergrundthread abgekoppelt. Der Entwickler muss dadurch bei der Implementierung der Observer sich keine Gedanken um Threadkollisionen machen. Der PTPHelper besitzt einen ***BackgroundHandler***, der im Konstruktor erstellt und an den UI Thread gekoppelt wird, siehe Codeausschnitt 3.10.

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

Codeausschnitt 3.10: 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. Der ***BackgroundHandler*** reagiert über die Methode ***handleMessage(Message)*** auf die Nachricht, indem jedoch die Methode auf dem UI Thread ausgeführt wird. Das ***SessionJoinRule*** Interface definiert die Methode ***canJoin(String joinersUniqueId)***. 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.
Schließlich gibt es noch den ***DataListener*** Interface, das für das Empfangen von Daten notwendig ist und die Methode ***dataSentToAllPeers(String,int,String[])*** definiert. Der erste String Parameter in der Methode ist die ***Unique ID*** des Versenders. Die beiden anderen Parameter können beliebige Daten beinhalten. Diese Methode wird jedesmal aufgerufen, wenn eine Nachricht an alle Smartphones geschickt wird. Die Implementierung und die Registierung des Listeners ist notwendig für das Empfangen von Daten. Die Abbildung 3.1 veranschaulicht den Nachrichtenablauf beim Versenden von Daten. Die Methode ***sendDataToAllPeers(int arg,String[] data)***, die beim PTPHelper implementiert ist, erlaubt es Daten an alle zu der Session verbundenen Spielern zu verschicken. Diese Methode sendet die Daten über den ***SignalEmitter*** an die anderen Spieler. Die ***SignalHandler*** reagieren auf diese Nachricht und leiten sie an die ***DataListener***, die vom Entwickler beim ***PTPHelper*** registiert wurden.

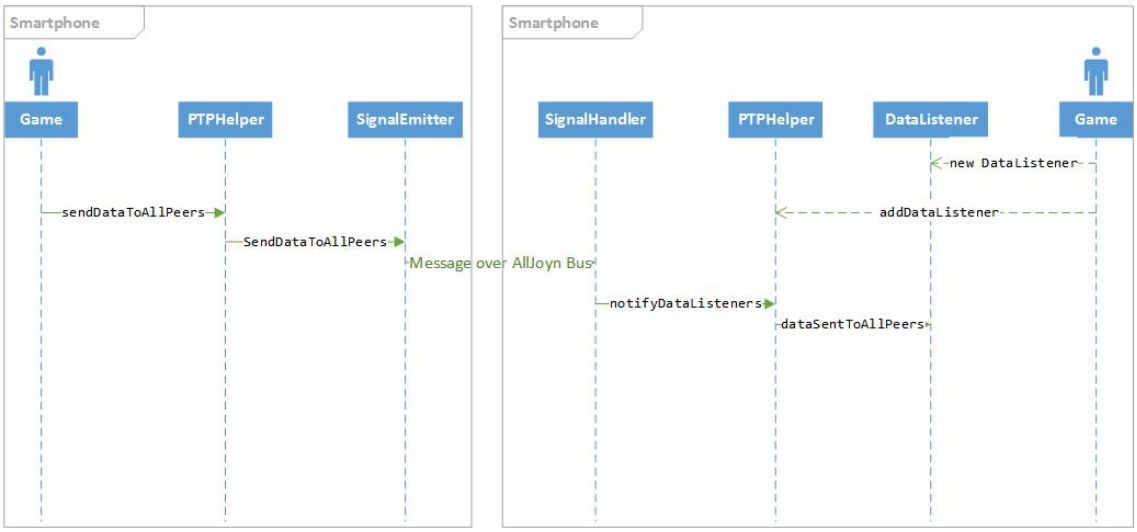


Abbildung 3.1: Nachrichtenablauf

AllJoyn baut auf einem ***DBus System*** auf und unterliegt somit dessen Specificationen[12]. Es ist daher zu beachten, dass der String, der verschickt wird, nur eine ***UTF-8*** Formatierung besitzen darf. Der SignalHandler wird beim PTPHelper erstellt, und er definiert eine Methode.

```
1 @BusSignalHandler(iface = "de.ptpservice.PTPBusObjectInterface", signal = "
      SendDataToAllPeers")
2 public void SendDataToAllPeers(String sentFrom,int arg, byte[] data) {
3   notifyDataListenersAllPeers(sentFrom,arg,data);
4 }
```

Codeausschnitt 3.11: SignalHandler Methode

Der Codeausschnitt 3.11 zeigt die Methode, die für die Kommunikation zwischen den Smartphones benutzt wird. Die Implementierung der Methode im SignalHandler, bedeutet dass diese Methode aufgerufen wird, wenn eine Nachricht an jedes Smartphone geschickt wird. In Zeile 3, werden die empfangenen Daten an die ***DataListener*** geschickt.

3.3 LobbyActivity

Die LobbyActivity ist eine abstracte Klasse, die eine Grundfunktionalität mitbringt. Die Lobby, die die LobbyActivity implementiert, zeigt die Abbildung 3.2. Sie bietet eine UI-Oberfläche, die 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 offene Spiele handelt. Für die Verbindung zu den Spiele, muss nur der entsprechende Name berührt werden. Die LobbyActivity ist eine abstrakte Klasse, daher müssen die Methoden ***getJoinSessionView*** und ***getHostSessionView*** imlementiert werden. Bei diesen Methoden handelt es sich um eine Definition der Klassen, die die Activities beschreiben, die geöffnet werden soll, wenn eine gegebene Aktion durchgeführt wird. Das heißt wenn ein neues Spiel erstellt wird, so muss die Methode ***getHostSessionView()*** die Activity Klasse definieren, die geöffnet werden soll, wenn ein Spiel geöffnet wird.Die beiden Methoden können auch die selbe Activity übergeben, wenn die Activity die logische Auseinandersetzung mit dem Host und Clienten übernimmt.

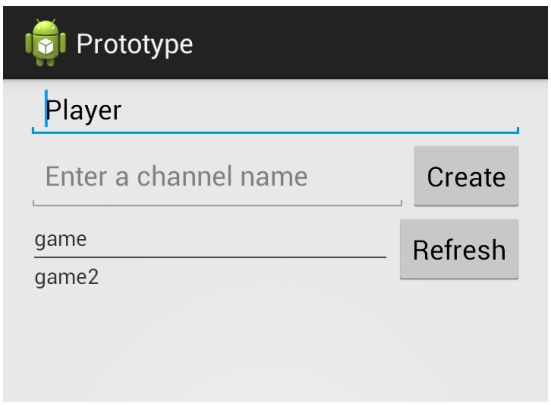


Abbildung 3.2: LobbyActivity

3.4 PTPService

Die PTPService Klasse besitzt die eigentliche Funktionalität der Bibliothek und ist dafür verantwortlich die Verbindung mit dem Bus zu erstellen und diese entsprechend zu konfigurieren. Der PTPService ist eine Unterklasse von Service aus dem Android SDK, die im Hintergrund gestartet wird und in einem seperaten Thread läuft. Der Service bietet keine direkten Zugriff auf die Instanz, und somit muss die Kommunikation mit einem Nachrichtensystem realisiert werden. Android bietet hierfür ein Nachrichtensystem an, um mit dem Service zu kommunizieren. Die Verwendung des Nachrichtensystems von Android beschränkt sich jedoch auf die Verwendung des ***Handlers***. 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

auszuführen. Damit das Bussystem von AllJoyn benutzt werden kann, muss dieser initialisiert werden. Das geschieht über einen Befehlsaufruf, der im Codeausschnitt 3.12 dargestellt ist.

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

Codeausschnitt 3.12: Daemon initialisieren

Der AllJoyn Daemon ist ein Hintergrundprozess, der sich um die verbindungstechnischen Abläufe kümmert. Daraufhin wird der HandlerThread gestartet, der sich um die Nachrichten von PTPHelper kümmert. Dabei wird wie beim PTPHelper ein BackgroundHandler initialisiert. Der BackgroundHandler wird jedoch an einen neu erstellten Thread gekoppelt. Der PTPHelper teilt über die Observer dem PTPService mit, welche Aktionen ausgeführt werden müssen.

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

3.4.1 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 3.13 zeigt wie das BusAttachement erstellt wird.

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

Codeausschnitt 3.13: BusAttachement erstellen

Dabei wird als erster Parameter der BusAttachement Name übergeben. Es ist möglicht mit AllJoyn mehrere Anwendungen auf den selben Smartphone laufen zu lassen. Um jedoch die einzelnen BusAttachements den Applicaiton zuweisen zu können, werden sie mit einem Namen versehen. Der zweite Parameter bestimmt, ob eingehende Nachrichten empfangen und behandelt oder verworfen werden sollen. Danach wird in der Methode der BusListener registriert, der dafür zuständig ist auf geöffnete Spiele zu reagieren. Diese Nachrichten werden dann an die Observer weiter geleitet, siehe Codeausschnitt 3.14.

```
@Override
public void foundAdvertisedName(String fullName, short transport, String
    namePrefix) {
  if(namePrefix.equals(packageName))
    PTPHelper.getInstance().notifyFoundAdvertisedName(getSimpleName(fullName));
};
@Override
public void lostAdvertisedName(String fullName, short transport, String
    namePrefix) {
  if(namePrefix.equals(packageName))
    PTPHelper.getInstance().notifyLostAdvertisedName(getSimpleName(fullName));
}
@Override
public void busDisconnected() {
  PTPHelper.getInstance().notifyBusDisconnected();
}
```

Codeausschnitt 3.14: BusListener Nachrichten

Der Packagename ändert sich innerhalb der Anwendung nicht, daher wird nur der Sessionname an die Observer weitergeleitet. Daraufhin wird der BusObject registriert, das im Codeausschnitt 3.15 zu sehen ist.

```
BusObject busObject = PTPHelper.getInstance().getBusObject();
status = bus.registerBusObject(busObject, objectPath+ "/" + getDeviceID());
```

Codeausschnitt 3.15: BusObject registrieren

Dabei wird ein BusObject genommen, der bei der Initialisierung vom PTPHelper erstellt wurde. Der BusObject wird später dazu benötigt einen sogenannte SignalEmmiter zu bekommen, über den Nachrichten an alle Smartphones geschickt werden kann. Der Pfad, unter den der BusObject abgelegt wird, muss innerhalb des BusSystems einzigartig sein, um Konflikte zwischen den BusObjekten zu vermeiden. Der Pfad ist wie bei einen Dateisystem organisiert und wird über das Slashzeichen getrennt. Um die Einzigartigkeit eines Pfaded für jedes Bus Objektes zu ermöglichen, wurde als Teilpfad die ***Geräte ID*** benutzt. Der Codeausschnitt 3.16 zeigt, wie die Geräte ID erfragt werden kann.

```
  TelephonyManager telephonyManager = (TelephonyManager) getBaseContext().
      getSystemService(Context.TELEPHONY_SERVICE);
  telephonyManager.getDeviceId();
```

Codeausschnitt 3.16: Geräte ID

Im AndroidManifest muss die notwendige Erlaubniss ***READ_PHONE_STATE*** hinzugefügt werden, um die Geräte ID zu bekommen. Darufhin wird die ***bus.connect()*** Methode aufgerufen, um mit dem Bus zu verbinden. Wenn die Verbindung erfolgreich war, wird der SignalHandler registriert. Schließlich wird die UniqueID des BusAttachements an den PTPHelper übergeben, die einen busweiten einzigartigen String darstellt. Die ***doDisconnect()*** Methode ist dafür verantworlich alle Handler und das BusObject abzumelden und die Verbindung zu trennen.

3.4.2 doStartDiscovery() und doStopDiscovery()

Die Methode ***doStartDiscovery()*** wird am Anfang, nach dem die Verbindung aufgebaut ist, ausgeführt. Die Methode ist dafür verantworlich dem BusAttachement mitzuteilen nach offene Spielen zu suchen und diese dem Observer mitzuteilen. Dies geschieht über den Methodenaufruf ***bus.findAdvertisedName(packageName)***. Der ***packageName*** stellt den Prefix für die den Session Namen dar. Das BusAttachement reagiert auf alle offene Spiele, die den Prefix mit dem ***packageName*** haben. Wenn das Suchen nach offenen Spielen beendet werden soll, kann die Methode ***doStopDiscovery()*** ausgeführt werden.

3.4.3 doBindSession() und doUnbindSession()

Die ***doBindSession()*** Methode wird dafür benötigt, um eine neue Session zu erstellen. Dabei werden unterschiedliche Konfigurationen vorgenommen, die zur Erstellung einer Session notwendig sin. Zum einen muss der ***SessionPort*** festgelegt und die Transportparameter konfiguriert werden. Bei den Transportparametern handelt es sich darum, ob die Verbindung über Bluetooth, WifiDirect oder Lan zugelassen werden soll, 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 soll. Die Konfigurationen sind im Codeausschnitt 3.17 aufgeführt.

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

Codeausschnitt 3.17: Sessionkonfiguration

Als ***contactPort*** wird der Standartwert 100 genommen. Dieser kann jedoch beim PTPHelper über die Methode ***setContactPort*** verändert werden. Neben den Konfi-

gurationsparametern 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 3.18:

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

Codeausschnitt 3.18: 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 3.19: Als Host den SignalEmitter bekommen

Die zweite Methode umfasst ein etwas komplizierteres 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 3.19 wird bis dahin unbekannte ***SessionID*** übergeben, da sie beim Binden der Session noch nicht bekannt war. Mit der ***SessionID*** wird ***SignalEmitter*** erstellt, 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 Spieler verbunden hat. Zum schluss wird der ***SessionPortListener*** beim Bus registriert und eine Session erstellt, siehe Codeausschnitt 3.20.

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

Codeausschnitt 3.20: Session erstellen

In der Methode ***doUnbindSession()*** wird durch den einfachen Methodenaufruf ***bus.unbindSessionPort(contactPort)***, die Session, die an den ***contactPort*** gebunden ist, geschlossen. Es ist noch anzumerken, dass der ***contactPort*** nicht mit dem Netzwerkport zu verwechseln ist, der in ***TCP*** oder ***UDP*** Verwendung findet. Der ***contactPort*** ist nur eine AllJoyn weite Nummer, die für das Erstellen von Sessions benötigt wird.

3.4.4 doRequestName() und doReleaseName()

Die Methoden ***doRequestName()*** und ***doReleaseName()*** sind dazu vorgesehen, um die Sessionnamen, die später durch das ***Advertising*** an andere Spieler mitgeteilt werden können, vom Bus zugewiesen zu bekommen. Es dürfen Sessionnamen nur dann an Andere mitgeteilt werden, wenn diese vom Bus auch zugewiesen worden sind. Der Codeausschnitt 3.21 zeigt den Methodenaufruf.

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

Codeausschnitt 3.21: Name für Session anfordern

Als Prefix der ***packgeName*** verwendet und der ***HostSessionName*** ist der Name, der 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.

3.4.5 doAdvertise() und doCancelAdvertise()

Das sogenannte ***Advertising*** wird dazu benutzt um anderen Spielern den Spielnamen mitzuteilen. Über einen einfachen Befehl wird die Mitteilung gestartet und beendet. Zur Verdeutlichung siehe Codeausschnitt 3.22.

```
1 bus.advertiseName(sessionName, SessionOpts.TRANSPORT_WLAN);
2  ...
3 bus.cancelAdvertiseName(sessionName,SessionOpts.TRANSPORT_WLAN);
```

Codeausschnitt 3.22: Session Name mitteilen

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

3.4.6 doJoinSession() und doLeaveSession()

Die Methode ***doJoinSession()*** wird von den beitretenden Teilnehmer ausgeführt. Im Codeausschnitt 3.23 zeigt die genauere Vorgehensweise beim Bestreten einer Session.

```
1 SessionOpts sessionOpts = new SessionOpts(SessionOpts.TRAFFIC_MESSAGES, true,
2 SessionOpts.PROXIMITY_ANY, SessionOpts.TRANSPORT_WLAN);
3 Mutable.IntegerValue sessionId = new Mutable.IntegerValue();
4 bus.joinSession(sessionName,contactPort,sessionId,sessionOpts,sessionListener);
```

Codeausschnitt 3.23: 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, der bei der Ausführung der Methode zugewiesen wird. Schließlich wird noch ein ***SessionListener*** als Parameter übergeben. Dieser Listener leitet die Nachrichten an die Observer, die im PTPHelper registriert sind, weiter. Jedoch wird die Information über die SessionId dem PTPHelper nicht mitgeteilt, da sie für den Entwickler, der den Observer gegebenenfalls implementiert, nicht von Interesse ist. Falls die Verbindung erfolgreich war wird anschließend ein ***SignalEmitter*** erstellt und dieser an den PTPHelper übergeben, wie im Codeausschnitt 3.24 aufgeführt.

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

Codeausschnitt 3.24: SignalEmitter bekommen als Client

3.4.7 doQuit()

Schließlich gibt es noch die ***doQuit()*** Methode, die aufgerufen wird, wenn der Service beendet werden soll. Der Codeausschnitt 3.25 zeigt den Inhalt dieser Methode.

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

Codeausschnitt 3.25: doQuit Methode

Es wird somit die Verbindung mit dem Bus geschlossen, falls sie noch offen ist. Dann wird der ***BusThread*** beendet, der 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 Graphenspiels basiert im darauf, dass die Spieler einen Graphen sehen, bei dem sich die Kanten anfangs überschneiden. Ziel des Spieles ist es die Knoten des Graphen so zu bewegen, dass sich keine Kante mehr überschneidet. Die Schwierigkeit beim Spiel ist, dass sich alle anderen Knoten mitbewegen. Lediglich die Knoten, die von den anderen Spielern gehalten bzw. bewegt werden, werden nicht mitbewegt. Durch diese Tatsache, lässt sich das Spiel nur mit mehreren Spielern gewinnen. Es handelt sich hierbei um ein sogenanntes kooperatives Mehrspielerspiel. Auf dem Bild 4.1 ist zu erkennen, dass der Knoten mit der Nummer 6 rot markiert ist. Somit kann der Spieler diesen Knoten nicht bewegen. Alle anderen Knoten, die blau markiert sind, können vom Spieler jedoch bewegt werden. Das Puzzel lässt sich einfacher lösen, wenn mehr Leute mitspielen. Dies bringt einen weiteren sozialen Aspekt, das das Spielen mit mehr Leuten fördert. Die Spieler können sich wärend des Spielens zu dem Spiel verbinden, d.h. selbst wenn ein Spiel schon im Gange ist, kann ein weiterer Spieler beitreten und mithelfen. Die Spieler können auch jederzeit das Spiel verlassen, ohne das Beenden des Spiels zu verursachen. Es ist jedoch zu beachten, dass immernoch mindestens 2 Spieler benötigt werden, um das Rätsel lösen zu können.

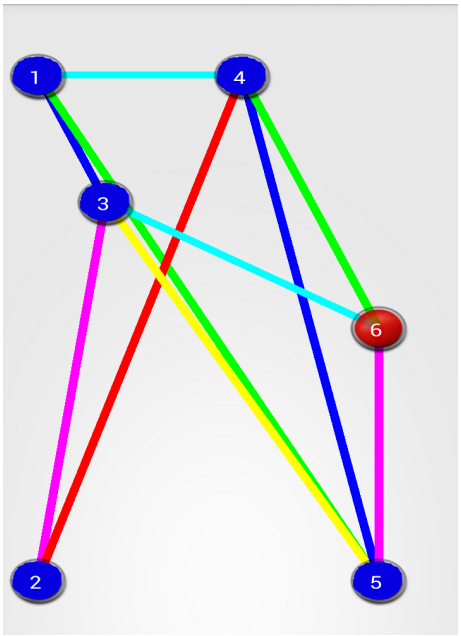


Abbildung 4.1: Graphspiel

4.2 Implementierung

4.2.1 Projektkonfiguration

Für die Entwicklung des Spieles ist das Eclipse ADT verwendet worden, sodass die Beschreibung der Projektkonfiguration sich auf Eclipse beruhen. Als Erstes wird ein ***Android Application Project*** erstellt. Dabei werden die Applicationname, Packagename und Projektname entsprechend angegeben. Eine vorimplementierte Activity wird entfernt, da eine eine Activity hierfür implementiert wird. Als das Target SDK wird ***Android 4.2 Jelly Bean*** und als Mininum wird ***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. Durch einen Rechtsklickt auf das Projekt erscheint ein Kontextmenu, bei den Eigenschaften ausgewählt werden. Unter den Reiter ***Android*** ist die Einstellung für Bibliotheken. Dort muss die ***PTPLibrary*** als Bibliothek angeben werden. Android erlaubt es nicht ***Libraries*** als gewöhnliche Jar Dateien zu packen, daher muss die Bibliothek als Android Projekt referenziert werden.[6]

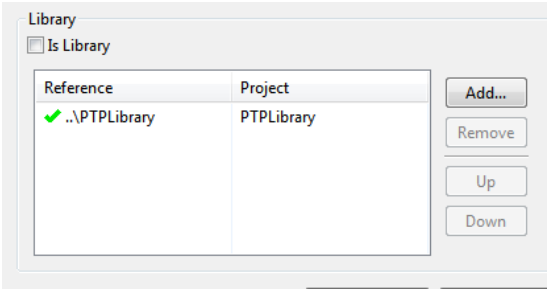


Abbildung 4.2: Bibliothekreferenz

4.2.2 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 wird die ***onCreate()*** Methode aufgerufen. Im Codeausschnitt 4.1 ist der Inhalt der Methode zu sehen.

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

Codeausschnitt 4.1: onCreate Methode

Dabei wird als erstes das Graphobjekt erstellt und konfiguriert, das das Datenmodel des Spieles representiert. Als zweites wird dann der ***PTPService*** initialisiert. Dabei wird die ***GraphLobbyActivity*** Klasse übergebe, die die ***LobbyActivity*** implementiert, übergeben. Die GraphLobbyActivity definiert nur die ***Activity***, die geöffnet werden soll, wenn eine Spielsession erstellt wurde, oder zu einer Spielsession verbunden wurde. Dann wird noch ein ***DataListener*** beim ***PTPHelper*** registiert, um auf Nachrichten von anderen Spielen reagieren zu können. Der Codeausschnitt 4.2 zeigt den Inhalt des ***DataListeners***.

```
1 PTPHelper.getInstance().addDataListener(new DataListener() {
2      @Override
3      public void dataSentToAllPeers(String peersID, int messageType, String[]
           data) {
4        MessageInfoHolder infoHolder = new MessageInfoHolder();
5        infoHolder.data = data;
6        infoHolder.sentBy = peersID;
7        sendMessage(messageType, infoHolder);
8      }
9    });
```

Codeausschnitt 4.2: DataListener

Der DataListener beinhaltet eine Methode, die die ID des Senders als String, den Nachrichtentyp als Integer, und die Daten als ein String Array übergibt. Diese Informationen werden als ein Object von Typ ***MessageInfoHolder*** zusammengefasst, um es an den ***MessageHandler*** als ***Message*** senden zu können. Der MessageHandler läuft auf dem ***UIThread*** und verhindert Threadkollisionen, da die Methode des DataListeners vom Thread des Hintergrundservices aufgerufen wird. Der MessageHandler entscheidet anhand des Nachrichtentyps die Methode, die aufgerufen werden soll, und ruft diese mit den Nachrichteninhalt auf. Im Codeausschnitt 4.3 sind die 2 Methoden aufgeführt, die beim Empfangen von Nachrichten verwendet werden.

```
1 private void nodeOwnerChanged(MessageInfoHolder obj) {
2   Node node = graph.getNodeFromXML(obj.data[0]);
3     graph.ChangeOwnerOfNode(node.getId(),node.getOwner(), obj.sentBy);
4 }
5
6 private void nodePositionChanged(MessageInfoHolder obj) {
7   Node node = graph.getNodeFromXML(obj.data[0]);
8     graph.MoveNode(node.getId(), node.getX(), node.getY(), obj.sentBy);
9 }
```

Codeausschnitt 4.3: Empfänger Methoden

Die Daten werden als ein XML String versendet und werden mithilfe einer Bibliothek namens ***PeerParser*** [13] in ein entsprechendes Object umgewandelt. Die Attribute des Objectes werden dazu verwenden Methoden auf dem Graph auszuführen.

4.2.3 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, die 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. Die im Codeausschnitt 4.4 dargestellten Methoden, sind die 2 Methoden, die für die Kommunikation mit anderen Spielern wichtig sind.

```
1 public void MoveNode(int id,double x,double y, String uniqueName);
2 public void ChangeOwnerOfNode(int id,String owner, String uniqueID);
```

Codeausschnitt 4.4: GraphInteface

Die Methode ***MoveNode*** wird aufgerufen, wenn ein Spieler einen Knoten verschoben hat. Die Methode ***ChangeOwnerOfNode*** wird aufgerufen, wenn jemand einen Knoten berührt oder loslässt, sodass der Knoten einem Besitzer zugewiesen werden kann. Der letzte Parameter ***uniqueID*** ist die ID von den Spieler, der diese Methode aufruft. Mithilfe dieses Parameters wird innerhalb der Methode entschieden, ob der Methodenaufruf lokal stattgefunden hat, oder von einem anderen Spieler stamm. Falls der Methodenaufruf lokal stattgefunden hat, werden die Daten über die Änderung in ein XML String gepackt und an die anderen Spieler verschickt. Der Codeausschnitt 4.5 zeigt einen Ausschnitt aus der Methode ***ChangeOwnerOfNode***

```
1   Node nodeToChange = new Node();
2   nodeToChange.setid(id);
3   nodeToChange.setOwner(owner);
4
5   XMLIdMap map=new XMLIdMap();
6   map.withCreator(new NodeCreator());
7   XMLEntity entity = map.encode(nodeToChange);
8   PTPHelper.getInstance().sendDataToAllPeers(NODE_OWNERSHIP_CHANGED, new String
        []{entity.toString()});
9   return;
10 }
```

Codeausschnitt 4.5: Sende Änderung des Besitzers

Dabei wird wieder die Bibliothek ***PeerParser*** verwenden um aus dem Node Object einen XML String zu erstellen. Dieser wird im Anschluss, mit Angabe des Nachrichtentyps ***NODE_OWNERSHIP_CHANGED***, über den PTPhelper an die anderen Spieler verschickt.

4.2.4 DrawView

Die ***DrawView*** ist für die grafische Darstellung und die Benutzereingaben zuständig. Diese Klasse ist die Unterklasse von ***View*** und implementiert zusätzlich den ***OnTouchListener*** und den ***GraphObserver***. Die ***View*** ist eine Android Klasse und beinhaltet als wichtigste Methode ***onDraw(Canvas)***, die dafür zuständig ist die Kanten und Knoten auf dem ***Canvas*** zu zeichnen. Außerdem wird in dieser Methode geprüft, ob der Graph gelöst ist und eine Meldung dass das Spiel gewonnen worden ist, 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, die sich um die Benutzeraktivitäten kümmert. 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. 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***, die auf die Nachricht hört, dass der Graph neu gezeichnet werden soll. Der Codeausschnitt 4.7 zeigt den Inhalt dieser Methode.

```
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. Diese Methode muss aufgerufen werden, wenn der Aufrufen nicht vom UI Thread kommt. Falls der Aufruf innerhalb des UI Threads befindet, wie z.B. bei der Abarbeitung der Berührungsaktivität reicht es die Methode ***invalidate()*** aufzurufen.

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 zu beachten sind. Um dieses Spiel spielen zu können, werden mindestens 2 Mitspieler und maximal 4 Spieler benötigt. Die maximal 4 Spieler beruhen auf der Tatsache, dass bei dieser Version ein Skatblatt benutzt wird, das nur 32 Karten umfasst. Am Anfang des Spiels, wenn alle Spieler sich zum Spiel verbunden haben, bekommt jeder 6 zufällige Karten ausgeteilt. Der Spieler, der das Spiel erstellt hat, darf anfangen. Dabei darf pro Zug, wenn der jeweilige Spieler an der Reihe ist eine Karte gespielt werden. 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, der 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

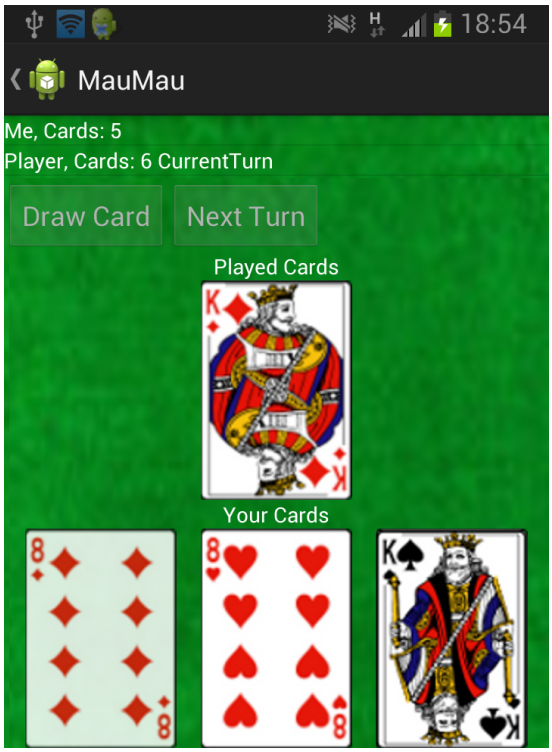


Abbildung 5.1: MauMau

lassen sich somit durch eine gewöhnliche Fingebewegung scrollen.
Die Regeln in dieser implementierten Version sind folgende:

- Es darf entweder die Spielfarbe oder die/den Zahl/Wert gespielt, von der Karte, die zuvor gespielt worden ist. Bezogen auf das Bild ***fig:maumau*** darft der nächste Spieler z.B. eine Karo 8 oder einen Pick König spielen.

- Falls davor eine Acht gespielt wurde, muss der nächste Spieler einen Zug aussetzen und der übernächste Spieler ist an der Reihe. Eine Ausnahme ist, wenn der auszusetzende Spieler eine 8 besitzt. Dann darf der Spieler diese Acht legen und der nächste Spieler muss aussetzen. Die gleiche Regel ist für den Ass anzuwenden.

- 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 schon 4 Karten ziehen, bis keiner mehr eine Sieben spielen kann.

- Falls ein Bube gespielt wurde, darf eine Spielfarbe gewählt werden, die von dem nächsten Spieler gespielt werden muss.

- Wenn eine Karte nicht gespielt werden kann, z.B. falls keine passende Karte vorhanden ist, muss eine Karte gezogen werden. Falls nach dem Ziehen immer noch keine passende Karte vorhanden ist, muss der Zug abgegeben werden und der nächste Spieler ist an der Reihe.

5.2 Anforderungen

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

- Es werden alle Spieler aufgelistet, die zum Spiel verbunden sind.
- Es wird angezeigt, wie viele Karten jeder Spieler besitzt.
- Jeder Mitspieler sieht die zuletzt gespielte Karte.
- Es dürfen nur die Karten gespielt werden, die vom Regelwerk her erlaubt sind.
- Ein Karte wird gespielt, indem das das Bild der Karte berührt wird.
- Wenn ein Bube gespielt wird, erscheint ein Dialog, bei dem die Spielfarbe gewählt werden 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 nicht von der im Kapitel 4.2.1 vorgestellten Konfiguration. Somit wird hier nicht mehr darauf eingegangen.

5.3.2 MauMauApplication

Die ***MauMauApplication*** Klasse erbt von ***Application*** und ist somit die Klasse, die 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", this, MauMauLobbyView.class);
```

Codeausschnitt 5.1: MauMau PTPHelper Initialisierung

Es werden der Applicationsname, der ***Context***, und die ***MauMauLobbyView*** Klasse übergeben. Der letzte Parameter ist die konkrete Implementierung der ***LobbyActivity***, die nur angibt, welche Views geöffnet werden sollen. Es wird außerdem noch ein ***DataListener*** beim ***PTPHelper*** registiert, um auf Nachrichten von anderen Spielen reagieren zu können. Der DataListener beinhaltet die gleich Implementierung, wie im Codeausschnitt 4.2 im Kapitel 4.2.2 dargestellt. Ein MessageHandler behandelt die empfangenen Nachrichten. Dieser führt, abhängig vom Nachrichtentyp, das in Form eines Integers mitgesendet wurde, die entsprechende Methode aus. Die 4 möglichen Methoden sind im Codeausschnitt 5.2 aufgeführt.

```
1 private void playerStateChanged(MessageInfoHolder message)
2 private void nextTurn(MessageInfoHolder message)
3 private void ownerChanged(MessageInfoHolder message)
4 private void cardPlayed(MessageInfoHolder message)
```

Codeausschnitt 5.2: Empfänger Methoden MauMau

Die Methode ***playerStateChanged*** wird aufgerufen, wenn ein neuer Spieler der Session begetreten ist, oder die Session verlassen hat. Der Inhalt der Nachricht ist zu der ID des Spielers auch der in der Lobby vergebene Name des Spielers. Wenn ein Spieler seinen Zug beendet hat, teilt er dies den anderen Mitspielern über die Methode ***nextTurn*** mit. Diese Nachricht beinhaltet die ID des Spielers, der den Zug beendet hat. Die Methode ***ownerChanged*** teilt allen Spielern mit, falls eine Karte den Besitzer gewechselt hat. Dabei wird zum einen die ID der Karte, sowie der neue Besitzer in Form eines XML String übermittelt. Schließlich gibt es noch die Methode ***cardPlayed***. Diese Methode wird von dem Spieler aufgerufen, der eine Karte spielt. Der Inhalt der Nachricht für diese Methode ist nur die ID der Karte, die gespielt wurde. Bei jeder dieser Methode wird auch die ID des Spielers übergeben, von dem die Nachricht stamm. Jede dieser Methoden ruft die entsprechende Methode auf dem ***GameManager*** auf.

5.3.3 GameManager

Der ***GameManager*** ist das Herzstück des Spiels und beinhaltet das Datenmodel und alle Informationen, die 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. Aus dem Wert der Karte wird ein Offset ermittelt und daraus wird die Position des Bildes dieser Karte berechnet. 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***, der 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 mindestens eine Regeln erfühlt sein muss. Ein Beispiel dafür ist die Regel, dass ein Spieler am Zug ist. Diese Regel muss immer gelten, wenn ein Spieler eine Karte spielen will, somit geht diese Regel in die inklusive Liste. Die Regeln, dass eine Spielfarbe oder ein bestimmter Wert gespielt werden muss, gehören in die exklusive Liste, da nur eine dieser Regeln erfühlt sein muss um eine Karte spielen zu können. Diese Liste wird 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*** besitzt 5 Methoden, die dazu bestimmt sind mit anderen Spieler zu kommunizieren. Diese Methoden werden entweder lokal von der ***View*** ausgeführt, oder von der ***MauMauApplication***, wenn der Aufruf von den anderen Spielern kommt. Die Methoden sind im Codeausschnitt 5.5 aufgelistet.

```
1  public void ChangeOwner(int cardId, String uniqueUserID);
2  public void PlayCard(int cardId, String uniqueUserID);
3  public void NextTurn(String uniqueUserID,int specialCase);
4  public void HiIAm(String uniqueID,String playerName);
5  public void ByeIWas(String uniqueID,String playerName);
```

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 am Zug ist und den Spielzug beendet hat. Ein Parameter ist immer die ***uniqueID***, sodass immer bei der Abarbeitung der Methoden klar ist, von wem der Methodenaufruf stamm. Die ***NextTurn*** Methode hat noch den zweiten Parameter ***specialCase***, der bestimmte Situationen kennzeichnet. Es wird z.B. nur die letzte Karte gespeichert, die gespielt wurde. Somit kann nicht mehr nachvollzogen werden, 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. Wenn ***specialCase=3*** ist, bedeutet 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, die allen Mitspielern mitteilt, welche Karte gespielt wurde und auf dem Bildschirm angezeigt werden kann.

5.3.4 GameActivity

Die ***GameActivity*** ist zum Einen für die Behandlung von Benutzereingaben zuständig und zum Anderen für die graphische Darstellung des Spiels. Die ***onCreate()*** Method beinhaltet die Initialisierung der Buttons und anderen UI Elementen wie z.B. der ***Gallery***. Die ***Gallery*** ist aus dem Android SDK und ist dafür vorgesehen mehrere Bilder darzustellen und stellt Funktionen bereit wie das Scrollen und Auswählen von diesen. Die Gallery ist jedoch seit de Android API 16 Deprecated. Als Ersatz kann auch die Klasse ***HorizontalScrollView*** verwendet werden. Um die ***Gallery*** mit Bildern zu füllen, wird ein ***BaseAdapter*** benötigt, das im Prinzip das Datenmodel darstellt. Damit die ***Gallery*** den aktuellen Stand der Karten anzeigt, wird der ***BaseAdapter*** upgedatet, das wiederum die Gallery benachrichtigt, die Karten neu anzuzeigen. Dann gibt es den ***OnItemClickListener***, der für die ***Gallery*** implementiert wird. Dieser Listener implementiert die Aktionen, die ausgeführt werden sollen, falls eine Karte berührt wird. Ein Auschnitt aus der ***OnItemClick*** Methode zeigt der 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 gespielt werden kann, indem es mit dem ***RuleEnforcer*** geprüft wird.

Danach wird geprüft, ob es sich bei der Karte um einen Buben handelt und ein Fenster angezeigt werden soll, das für das Auswählen einer Wunschfarbe vorgesehen ist. Dieses Fenster ist auf dem Bild 5.2 zu sehen. Andernfalls wird die Karte gespielt, in dem die Karten-ID dem ***GameManager*** mitgeteilt wird. Die ***GameActivity*** implementiert auch den ***GameManagerObserver*** Inteface, sodass es auch eine ***update(int)*** Methode

Abbildung 5.2: Spielfarbe wünschen

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 gezeichnet werden muss. Es gibt bei Android auch die Möglichkeit auf den Zurückbutton zu drücken,was dafür sorgt,dass man die vorheriege View aufgerufen wird. Dies wird dem ***GameManager*** und dem ***PTPService*** mitgeteilt, da durch das drücken des Zurückbuttons das laufende Spiel verlassen wird. Der Codeausschnitt 5.7 zeigt die Behandlung dieses Zurückbuttons.

```
1 @Override
2 public void onBackPressed(){
3   super.onBackPressed();
4   gameManager.reset();
5   PTPHelper.getInstance().leaveSession();
6   PTPHelper.getInstance().disconnect();
7   PTPHelper.getInstance().connectAndStartDiscover();
8 }
```

Codeausschnitt 5.7: Behandlung des Zurückbutton

Dabei wird die ***onBackPressed()*** Methode von der ***Activity*** überschrieben. Weiterhin wird der ***gameManager*** zurückgesetzt, sodass alles wieder mit Anfangswerten initialisiert ist und ein neues Spiel gestartet werden kann. Zum Schluss wird dem ***PTPHelper*** mitgeteilt die laufende Session zu schliessen und eine neue Verbindung aufzubauen, sodass der Anfangszustand erreicht wird, da durch die Zurücktaste die Lobby aufgerufen wird.

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, die durch den größeren Synchronizationsaufwand entstehen. Besonder an dem Praktischen Beispiel Mau-Mau wird deutlich, dass es viele Fälle gibt, die zu beachten sind, um den Zustand synchron zu halten. Daher war es notwendig mehr Informationen über das Netzwerk zu verschicken, das wiederum zu mehr Datenverkehr führt. Das Kartenspiel ist jedoch ein Rundenbasiertes Spiel, sodass eine größere Datenmenge das Spielerlebniss nicht beeinträchtigt. Das Rahmenwerk AllJoyn hielt was es versprach, mit all seinen Funktionalitäten, bis auf den leichten Einstieg. Das Verstandniss des Mechanismus ist anfangs recht mühsam und bedarf viel Ziel. Eine weitere Problematik kommt aus der Tatsache, dass der Android Emulator, der 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 Smartphones 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, das jedoch einen großen Aufwand bedeutet und die Auseinandersetzung mit dem Android Quellcode. Es gibt noch viel Entwicklungspotential für die hier vorgestellte Bibliothek, sowie das verwendete Rahmenwerk AllJoyn. Es fehlt z.B. die Möglichkeit ein Spiel über die Bluetooth Verbindung zu spielen. AllJoyn bietet eine Bluetooth Unterstützung nur für Smartphones mit ***root***[14] Rechten. Weiterhin wäre eine automatische Erstellung eines AccessPoints zu den sich die Spieler auch automatisch verbinden ein interessantes Feature. Weiterhin wäre die Entwicklung der Bibliothek auch für das iOS möglich, sodass auch

die Entwicklung von Peer-to-Peer Spielen für das iPhone und iPad erleichtert wird. Als eine Opensource Alternative zu Samsungs Chord SDK, hat diese Bibliothek das Potential, bei der Entwickung von Peer-to-Peer Spielen auf dem Android, einen großen Beitrag zu leisten.

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]

[10] Latenzzeiten von HSDPA http://www.elektronik-kompendium.de/sites/kom/0910251.htm [Stand: 18.07.2013]

[11] Netzabdeckung von Mobilfunkanbietern http://www.welt.de/wirtschaft/webwelt/article110550438/Schlecht-verbunden-mit-O2-und-E-Plus.html [Stand: 22.07.2013]

[12] DBus Datentypspecification http://dbus.freedesktop.org/doc/dbus-specification.html [Stand: 03.07.2013]

[13] PeerParser Bibliothek von der Universität Kassel

[14] Android Smartphone root Rechte http://google.about.com/od/socialtoolsfromgoogle/a/root-android-decision.htm