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

Lokales Wlan baisertes Multiplayer Spieleframework 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 eines Rahmenwerks für Mehrspielerspiele auf Basis des P2P-Rahmenwerks Alljoyn und zwei weiteren Spielen, welche mithilfe des Rahmenwerks erstellt worden sind und als praktische Beispiele dienen. Das Rahmenwerk wird auf Basis des Android SDK realisiert und sollte eine Kommunikation zwischen mehreren Geräten soweit vereinfachen, dass man sich bei der Implementierung weiterer Spiele nur um die Spielmechanik Gedanken machen muss und die Lobby-Funktionalität, wie das Erstellen und das Verbinden zum Spiel vom Rahmenwerk übernommen wird.

Als weiteres Kriterium gilt die Fähigkeit, Spiele im lokalen Wlan sichtbar zu machen und so eine Mehrspielerpartie zu ermöglichen ohne einen dedizierten externen Server, sowie einer Internetverbindung zu benötigen. Es sollte zum Spielen nur mindestens zwei Androidgeräte benötigen, die sich im selben Netzwerk befinden.

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

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

1 Einleitung

Spiele auf mobilen Endgeräten gibt es unzählige. Jedoch sind die meisten dieser Spiele entweder Einzelspielerspiele oder sind Serverbasierte Mehrspielerspiele, deren Server nur über eine aktive Internetverbindung erreicht werden kann. Trotz der mittlerweile gut ausgebauten Netzabdeckung und guter Internetgeschwindigkeit auf Smartphones, kann das Spielerlebniss durch eine schwankende Internetgeschwindigkeit getrübt werden, vorallem dann wenn eine volle 3G Geschwindigkeit nicht garantiert ist. So sind z.B. Actionspiele meist nur über eine Hotspot 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. Gerade in solchen Situationen ist die Möglichkeit miteinander eine Mehrspielerpartie zu starten ohne eine Internetverbindung zu haben sehr willkommen. Um die Entwicklung solcher Spiele vorran zu treiben und es den Entwickler soweit wie möglich zu erleichtern, sollte im Rahmen dieser Arbeit ein Rahmenwerk entwickelt werden, welches im Folgenden als PTP-Rahmenwerk genannt wird. Das PTP steht als Abkürzung von Peer-to-Peer, welches als Prinzip noch genauer erklärt wird. Es sollen die kommunikatonsbedingen Herausforderungen mit dem PTP-Rahmenwerk 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 und die Übertragen der Daten an die einzelnen Gerät würde sozusagen vom PTP-Rahmenwerk übernommen. Damit die Entwicklung solcher Spiele für die Hobbyentwickler keine finanziellen Hürden stellt, wurde das PTP-Rahmenwerk mit dem Android SDK 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 das PTP-Rahmenwerk auch für praktische Anwendungen, wie z.B. Textmessanger, verwenden, jedoch wird im Rahmen dieser Ausarbeitung nur auf die Realisierung von Spielen eingegangen.

1.1 Ziel und Aufgabenstellung der Arbeit

Ziel dieser Arbeit ist es die Lokalisierung und Verbindung der Geräte vom PTP-Rahmenwerk übernehmen zu lassen. Somit hat der Entwickler sich nur um die Implementierung der Schnittstellen-Objekte zu kümmern über die die Kommunikation abläuft. Das PTP-Rahmenwerk soll auf dem Android SDK aufbauen, da es sich hierbei um eine kostenlos verfügbaren Entwicklungsframework handelt. Weiterhin sind zwei Spiele als praktische Beispiele mithilfe dieses PTP-Rahmenwerkes zu implementierung um zum einen die Vorgehensweise zu demonstrieren und zum Anderen die Realisierbarkeit zu testen.

1.2 Lösungsweg der Aufgabenstellung

Um sich die ganzen Herausforderung bei der Realisierung Endgerät-zu-Endgerät 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 das PTP-Rahmenwerk, weiterhin mussten die vielen, gerade auf den ersten Blick komplizierten, Konfigurationen des AllJoyn Rahmenwerks durch eine einfache Schnittstelle erweitert werden. Außerdem musste ein Multithreaded Handler-System verwendet werden um UI-Prozesse des Spiels nicht durch die Hintergrundprozesse der Gerätekommunikation zu behindern. Als praktische Anwendungsbeispiele wurde ein Echtzeitpuzzlespiel und ein Turnbased Kartenspiel namens MauMau implementiert.

1.3 Gliederung

Die Arbeit beinhaltet zu Anfang die Erklärung der verwendeten Rahmenwerke AndroidSDK und AllJoyn. Dabei werden die Konzepte dieser Rahmenwerke grob beschrieben. Als nächstes wird das Konzept und die Impelementierung des Multiplayer Spieleframeworks, welches das Hauptaugenmerk dieser Arbeit ist, beschrieben. Daraufhin wird beschrieben wie die zwei Spielebeispiele realisiert wurden, sowie Herausforderung, die während der Implementierung entstanden. Zum Schluss gibt es noch eine theoretische Auseinandersetzung mit dem Prinzip der Wlan basierten P2P-Spiele.

2 Grundlagen

2.1 Konzepte

Das PTP-Rahmenwerk basiert auf dem Peer-to-Peer Prinzip und benötigt deswegen keinen weiteren Server. Weil durch einen Serverbasierten Ansatz der Zustand des Spiels nur auf dem Server organisiert wird ist es eine gewisse Herausforderung bei einem Peer-to-Peer Ansatz alle Gerät synchron zu halten. Der Entwickler hat nur Objekte, über die er die Kommunikation realisieren muss. Dabei handelt es sich zum einen um Objekte, die die eingehenden Signale behandeln und zum Anderen Objekte um Nachrichten an die anderen Spieler zu senden.

Jedes der Geräte hat zum einen Handler sowie einen Emitter. Der Emitter ist das Objekt welches die Signale an andere Geräte rausschickt und der Handler ist somit das Objekt, welches auf jedem Gerät die eingehende Nachricht behandelt. In den meisten Anwendungsfällen haben die Emitter und Handler auf allen Geräten die selbe Impelementierung. Zusätzlich wird eine Session verwendung um die einzelnen Spielsitzungen oder Spielrunden in sich geschlossen zu handhaben. Somit erstellt ein Host eine Session zu der sich die die Klienten verbinden können um an der Spielrunde teilnehmen zu können. Dadurch wird außerdem die Möglichkeit gegeben mehrere Spiele simultan im selben Netzwerk zu spielen.

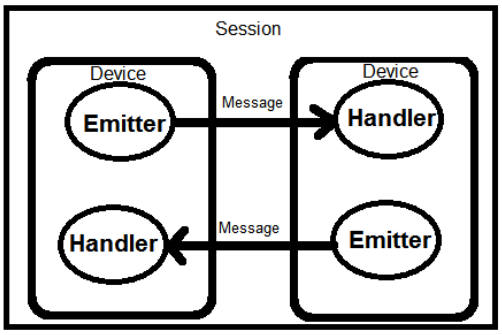

Abbildung 2.1: Kommunications-Konzept

2.2 Peer-to-Peer

Peer ist das englische Wort für Gleichstehender oder Gleichberechtigter, somit ist das Prinzip eines Peer-to-Peer Netzwerk, dass alle Teilnehmer gleichberechtigt sind. 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.

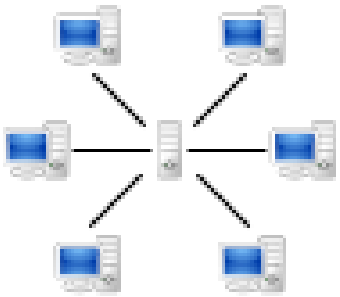

Abbildung 2.2: Server basiertes Netzwerk

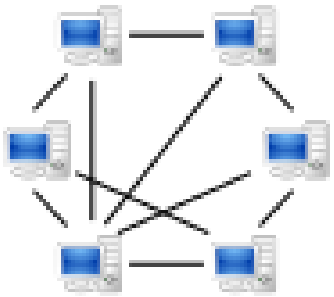

Abbildung 2.3: PtP basiertes Netzwerk

Die Bilder 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 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. Da die Berechnungen meistens auf dem Server liegen werden die verbundenen Geräte entlastet. 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 um zu garantieren, dass alle den selben Zustand haben. 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 frei 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.

Abbildung 2.4: Android

Somit waren im März 2013 etwa 750 Millionen Android End-Geräte aktiviert und man merkt schnell, dass die Popularität dieses Betriebssystem immer mehr zunimmt. Da ein Smartphone Betriebsystem auch von den angeboten Apps lebt, hat Google eine Entwicklungswerkzeugsammlung zur Verfügung gestellt, welche die Entwicklung von Applikationen für Android möglichst einfach ermöglichen 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 und dank Eclipse zu einer höheren Produktivität beiträgt. Als nächstes gehe ich auf die einzelne Grundlagen von Android SDK ein 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. Ein Use-Case bei dem man mehrere Fenster hat, würde sozusagen mehrere Activities nacheinander aufrufen. Um eine Activity zu erstellen muss seine Klasse eine Unterklasse von Activity sein. Zusätzlich muss man die Methode ***onCreate()*** in seiner Klasse überschreiben. Diese Methode wird jedesmal aufgerufen wenn die Activity erstellt wird, also auf dem Bildschirm erscheinen soll. In diese Methode kommen Aufrufe von Fenstern, 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 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.

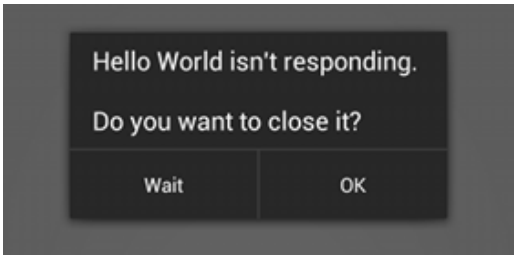

Abbildung 2.5: ANR

Somit ist es wichtig langwierige und blockende Aufgaben wie die Netzwerkkommunikation in ein seperaten Thread auszulagern um den UI Thread nicht zu überlasten Denn ein blockierter UI Thread kann schnell zu einer sogenannten ANR-Meldung führen oder auf deutschen Geräten 'Anwendung reagiert nicht mehr'. Außerdem muss man bei Verwendung von mehreren Thread darauf achten, dass die UI Komponenten nur vom UI Thread angefasst werden dürfen. Somit muss eventuell bei Background Thread ***runOnUiThread()*** Methode verwendet werden, welche die auszuführende Aufgabe an den UI Thread übergibt.

2.3.7 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. So müssen alle Berechtigungen z.B. folgendermaßen angegeben werden:

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

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

Abbildung 2.6: 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 mit einander 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. 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. Jedes dieser Anwendung muss einen sogenannten BusAttachement erstellen und den entsprechenden EventHandler implementieren über den dann die Nachrichten von anderen Anwendungen behandelt werden. Über einen ProxyObjekt lassen sich dann entweder Methoden auf einem bestimmten Gerät aufrufen oder auf allen Geräten, die zu dem Bus verbunden sind und das entsprechende Interface implementieren.

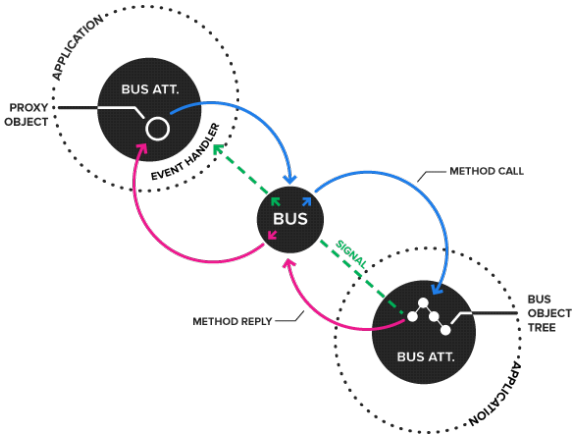

Abbildung 2.7: Bussystem

2.4.2 BusAttachement

Um mit dem Bussystem zu kommunizieren, erstellt man einen BusAttachement:

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

Der “applicationName” Parameter ist notwendig um verschiedene BusAttachements zu den entsprechenden Anwendungen zu zuordnet, falls mehrere Anwendungen über den selben Bus kommunizieren. Als Zweitparameter lässt sich noch angeben ob das BusAttachement eingehende Nachrichten verwerfen oder behandeln soll.

2.4.3 Interface

Die Kommunikation bei AllJoyn funktioniert über Interfaces, also eine konkrete Beschreibung der Methoden über die Nachrichten verschickt werden. So muss zum einen über dem Interfacenamen die folgende Annotation stehen:

```
1 @BusInterface (name = "de.package.MyInterface")
2 public interface MyInterface{}
```

Das Interface wird benötigt um über das BusAttachement an die ProxyObjekt zu kommen. Weiterhin muss das Inteface auch die Methoden entsprechend deklarieren.

```
1 @BusMethod
2 public void MyMethod() throws BusException;
```

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.

2.4.5 BusObject

Ein BusObject ist ein Objekt welches das definierte Inteface implementiert und unter einem bestimmten Pfad abgespeichert wird.

```
bus.registerBusObject(busObject, "/objectPath");
```

Somit können alle Anwendungen über den Pfad auf das Bus-Objekt zugreifen und darauf Methoden aufrufen oder Properties abfragen.

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 und bei SignalHandler handelt es sich um Objecte die auf Nachrichten reagieren die an alle Geräte geschickt werden wie bei einem Broadcast. Falls es sich bei dem Inteface um einen SignalHandler handelt müssen die Methoden @BusSignal als Annotationen beinhalten

```
@BusSignal
public int MyHandler() throws BusException;
```

Zusätzlich muss man bei der Implementierung eine weitere Annotation hinzufügen um den Nachrichtentyp explizit zu definieren.

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

2.4.7 SignalEmitter

Ein SignalEmitter ist das Objekt das dazu verwendet wird Nachrichten an alle Teilnehmer zu senden. Der SignalEmitter kann dazu verwendet werden um an das Inteface zu kommen welches auch von den SignalHandlern implementiert wird. Es ist im Prinzip der Proxy zu allen Teilnehmern. Den SignalEmitter bekommen man durch den folgenden Aufruf:

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

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.

2.4.8 Session

Eine Session ermöglicht es mehrere Teilnehmer zu einer gemeinsamen Einheit zusammenzufassen um verschiedene Anwendungsabläufe wie z.B. eine Spielrunde von einander getrennt zu handhaben. Dazu muss z.B. der Host einen sogenannten Channel erstellen, 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, welche sich im Bussystem befinden,über das sogennante Advertising mitteilen. Um diese Mitteilungen auch zu empfangen, müssen die Teilnehmer einen BusListener implementieren, welcher dann auf solche Nachrichten horcht. Somit hat einer der Teilnehmer die Aufgabe des Hosts und erstellt einen Channel und teilt diesen dann den Anderen mit:

```
1 bus.bindSessionPort(contactPort, sessionOpts, sessionPortListener);
2 bus.requestName("myChannelName");
3 bus.advertiseName("myChannelName");
```

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 wären z.B.

```
1 public boolean acceptSessionJoiner(short sessionPort,
2                String joiner, SessionOpts sessionOpts) {}
3
4 public void sessionJoined(short sessionPort, int id, String joiner) {}
```

Der Host braucht dann die Implementierung der Methoden um das Verbinden zu der Session zu Kontrollieren.

3 Implementierung

Es gibt viele verschiedene Spielprinzipien und daher sollte soweit es geht mit dem PTPRahmenwerk 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 das PTPRahmenwerk 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 das PTPRahmenwerk übernehmen. Um das Framework benutzen zu können müsste der Entwickler dann nur noch eine Jar-Datei 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 den Helper realisieren und erst bei Multiplayermodus den Service starten. Um den PTPHelper zu initialisieren muss der Entwickler in seinem Code den Folgenden Befehl ausführen:

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

Daraufhin kann er über den Getter an die initialisierte Instanz kommen.

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

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 erfragen kann man über die Methode

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

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. Dies geschieht über den folgenden Befehlsaufruf:

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

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.

Das PTPRahmenwerk 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.

```
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   //Show there is no WiFi-Connection message
5 }
```

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:

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

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:

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

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();
```

Dadurch kann man zusätzlich noch auf das Öffnen und Schließen der Spiele reagieren, falls die im PTPRahmenwerk vorimplementierte Behandlung, für den Entwickler nicht ausreichend ist. 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. Die zu implementierenden Methoden sind:

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

Dies ermöglicht es dem Entwickler bestimmte Aktionen auszuführen, falls neue Spieler sich zum Host verbinden oder sich vom Host trennen. 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

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

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

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

Ein entsprechend umfangreicheres Beispiel dafür ist die Methode

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

Diese Methode hat nämlich viele solcher Aktionen zusammengefasst.

```
public synchronized void hostStartChannel() {
  notifyHelperObservers(PTPService.UNBIND_SESSION);
  notifyHelperObservers(PTPService.RELEASE_NAME);
  notifyHelperObservers(PTPService.BIND_SESSION);
  notifyHelperObservers(PTPService.REQUEST_NAME);
  notifyHelperObservers(PTPService.ADVERTISE);
}
```

Diese Methode sorgt zum einen dafür, dass ein bereits erstellter Channel geschlossen wird um Konflikte zu vermeiden. Daraufhin wird ein neuer Channel erstellt, ein Channelname angefragt und zum Schluss über das Netzwerk allen Teilnehmern mitgeteilt. Die oben genannten Observer sind mithilfe eine ***Handlers*** vom Hintergrundthread abgekoppelt. Dies verhindert, dass der Entwickler bei der Implementierung der Observer sich gedanken um Threadkollisionen machen muss. Der PTPHelper besitzt nämlich einen ***BackgroundHandler***, welcher im Konstruktor erstellt und an den UI Thread gekoppelt wird.

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

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.

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

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

```
public void handleMessage(Message msg) {
  switch (msg.what) {
    ...
    case MEMBER_JOINED: notifyMemberJoined((String) msg.obj); break;
    ...
  }
}
...
private void notifyMemberJoined(String uniqueId) {
  for (SessionObserver obs : sessionObservers) {
        obs.memberJoined(uniqueId);
    }
```

```
13 }
```

Dann gibt es noch das SessionJoinRule Interface, welches die folgende Methode definiert.

```
1 public boolean 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.

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 folgenden zwei Methoden implementieren.

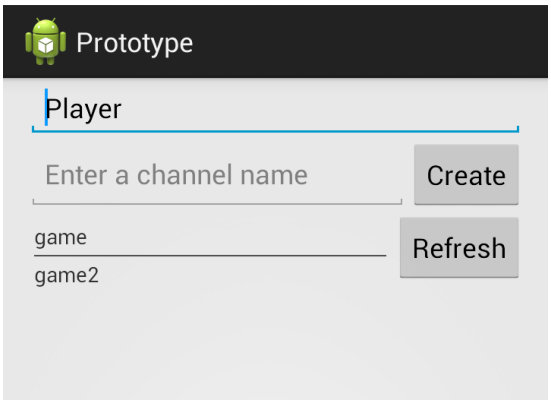

Abbildung 3.1: LobbyActivity

```
1  protected Class<?> getJoinChannelView();
2  protected Class<?> getHostChannelView();
```

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 Activity 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 entsprechenden Methoden representiert:

```
1 public void refresh(View view);
2 public void create(View view);
```

Die Methode ***create*** z.B. überprüft zuerst ob der Spielername und der Spielname angegeben sind und startet darauf die Session über die folgenden Befehle:

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

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. Da die Verbindung ein asynchroner Prozess ist muss über den Observer auf den Verbindungsstatuswechsel reagiert werden. Dies geschieht durch die folgende Implementierung:

```
1  public void connectionStateChanged(final int connectionState){
2    updateUIState(connectionState);
3    if(connectionState == PTPHelper.SESSION_HOSTED){
4      Intent intent = new Intent(LobbyActivity.this, getHostChannelView());
5      LobbyActivity.this.startActivity(intent);
6    }
7    if(connectionState == PTPHelper.SESSION_JOINED){
8      Intent intent = new Intent(LobbyActivity.this, getJoinChannelView());
9      LobbyActivity.this.startActivity(intent);
10   }
11 }
```

Somit wird die View oder die Activity des Spiels erst geöffnet, wenn vom Hintergrundservice die Benachrichtigung kommt, dass die Session erstellt ist oder einer Session begetreten wurde.
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 des Rahmenwerks und ist dafür verantwortlich die Verbindung mit dem Bus zu erstellen und entsprechen 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 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());
```

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. Dies geschieht über die folgende Methode:

```
1 private void startBusThread() {
2   HandlerThread busThread = new HandlerThread("BackgroundHandler");
3   busThread.start();
4   backgroundHandler = new BackgroundHandler(busThread.getLooper());
5 }
```

Der BackgroundHandler ist eine Unterklasse vom Handler aus dem Android SDK, welcher dafür genutzt wird Aufgaben an andere Threads zu übergeben. So hat der BackgroundHandler die folgenden Methoden implementiert:

```
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() {}
```

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. Ein Beispiel für die Implementierung einer dieser Methoden ist die Methode ***connect()*** :

```
1  public void connect() {
2      Message msg = backgroundHandler.obtainMessage(CONNECT);
3      backgroundHandler.sendMessage(msg);
4  }
5
6  ...
7
8  public void handleMessage(Message msg) {
9        switch (msg.what) {
10       case CONNECT:
11              doConnect();
12              break;
13           case DISCONNECT:
14             doDisconnect();
15           break;
16          ....
17 }
```

Die weitere Methode ***handleMessage(Message msg)*** implementiert die Behandlung der Nachricht vom anderen Thread. Somit führt die Nachricht ***CONNECT*** dazu, dass der BusThread die Methode ***doConnect()*** ausführt. Die Methoden vom BackgroundHandler werden wiederum über die Methode ***doAction(int)*** ausgeführt, welche durch den HelperObserver definiert ist. Diese Methode ist folgendermaßen implementiert:

```
public void doAction(int arg){
  switch(arg){
    case CONNECT: backgroundHandler.connect();break;
    case DISCONNECT: backgroundHandler.disconnect();break;
    ....
}
```

Als nächstes wird auf die einzelnen Methoden eingegangen die vom BusThread 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. Das BusAttachement wird folgendermaßen erstellt:

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

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.

```
@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();
}
```

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 über die folgenden Befehle geschieht:

```
Status status;
BusObject busObject = PTPHelper.getInstance().getBusObject();
if(busObject != null){
  status = bus.registerBusObject(busObject, objectPath+ "/" + getDeviceID());
  if (Status.OK != status) {
    Log.e(TAG, "Cannot register : " + status);
    return;
  }
}
```

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

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

Jedoch muss man dazu noch in dem AndroidManifest die notwendige Erlaubniss hinzufügen, nämlich:

```
    <uses-permission android:name="android.permission.READ_PHONE_STATE" >
    </uses-permission>
```

Darufhin 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.

```
bus.registerSignalHandlers(PTPHelper.getInstance().getSignalHandler());
```

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.

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

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

```
bus.unregisterBusListener(busListener);
bus.unregisterBusObject(PTPHelper.getInstance().getBusObject());
bus.unregisterSignalHandlers(PTPHelper.getInstance().getSignalHandler());
bus.disconnect();
```

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:

```
Status status = 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:

```
bus.cancelFindAdvertisedName(packageName);
```

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.

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

Als ***contactPort*** wird als 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 zum einen die folgende Methode:

```
1 public boolean acceptSessionJoiner(short sessionPort,
2          String joiner, SessionOpts sessionOpts) {
3     if(sessionPort == contactPort){
4       return PTPHelper.getInstance().canJoin(joiner);
5     }
6     return false;
7 }
```

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. 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:

```
1 public void sessionJoined(short sessionPort, int sessionId,String joiner) {
2     if(firstJoiner){
```

```
3       firstJoiner = false;
4       hostSessionId = sessionId;
5       SignalEmitter emitter = new SignalEmitter(PTPHelper.getInstance().
            getBusObject(), sessionId,
6           SignalEmitter.GlobalBroadcast.Off);
7       hostInterface = emitter
8           .getInterface(PTPHelper.getInstance().getBusObjectInterfaceType());
9       PTPHelper.getInstance().setSignalEmiter(hostInterface);
10      PTPHelper.getInstance().notifyMemberJoined(joiner);
11      bus.setSessionListener(sessionId, new PTPSessionListener());
12    }
13  }
```

Durch diese Methode 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. Anschließend wird der ***SessionPortListener*** beim Bus registriert:

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

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

```
1  bus.unbindSessionPort(contactPort);
```

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. Dies geschieht über den folgenden Aufruf:

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

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.

```
1 bus.releaseName(PTPHelper.getInstance().getHostChannelName());
```

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:

```
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);
```

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 dem beitretenden Teilnehmer ausgeführt. Wie schon erwähnt ist das Beitreten zu einer Session für den Host nicht zulässig, somit wird in diesen Fall die Methode frühzeitig verlassen.

```
1 if (hostChannelState != HostChannelState.IDLE) {
2   if (joinedToSelf) {
3     return;
4   }
5 }
```

Für einen normalen Teilnehmer wird der folgende Code ausgeführt:

```
String wellKnownName = packageName + "." + PTPHelper.getInstance().getChannelName
    ();
SessionOpts sessionOpts = new SessionOpts(SessionOpts.TRAFFIC_MESSAGES,
    true, SessionOpts.PROXIMITY_ANY, SessionOpts.TRANSPORT_WLAN);

Mutable.IntegerValue sessionId = new Mutable.IntegerValue();

bus.joinSession(wellKnownName, contactPort, sessionId,sessionOpts,
    sessionListener);
```

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. Dieser ist dafür verantwortlich Aktivitäten in der Session an den PTPHelper mitzuteilen, was über die folgenden Methoden geschieht:

```
public void sessionLost(int sessionId) {
  PTPHelper.getInstance().notifySessionLost();
}

public void sessionMemberAdded(int sessionId, String uniqueName) {
  PTPHelper.getInstance().notifyMemberJoined(uniqueName);
}

public void sessionMemberRemoved(int sessionId, String uniqueName) {
  PTPHelper.getInstance().notifyMemberLeft(uniqueName);
}
```

Die Information über die SessionId wird 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.

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

3.3.7 doQuit()

Schließlich gibt es noch die ***doQuit()*** Methode, die aufgerufen wird, den der Service beendet werden soll. Diese sieht wie folgt aus:

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

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

5 Praktisches Anwendungsbeispiel: MauMau

6 Herausforderungen und Probleme

7 Fazit

A Abkürzungsverzeichnis

B Literaturverzeichnis

[Mus09] Mustermann, Max: *Titel. Untertitel.* Auflage. Verlagsort: Verlag, Jahreszahl (= Reihe).

[Lor09] Ebers, Robin (2009): „Lorem Ipsum“. URL: `http://de.lipsum.com` [Stand: 11.10.2009]