Hochschule
Kaiserslautern
University of
Applied Sciences

— Projektbericht —

BTInvoke: Ausführen von Java Methoden über Bluetooth

Patrick Schwartz

16. Februar 2015

Hochschule Kaiserslautern
Fachbereich Informatik und Mikrosystemtechnik
Master Informatik

Mobile Anwendungen mit Android
bei Prof. Dr. Manh Tien Tran

Inhaltsverzeichnis

1 Einleitung

Dieses Kapitel dient als Einleitung dieses Projektberichtes. Der erste Abschnitt beschreibt kurz, um was für ein Projekt es sich handelt. Der nächste Abschnitt geht auf Voraussetzungen und Anmerkungen zu dem Projekt ein. Im letzten Abschnitt wird der Aufbau dieses Dokumentes erläutert.

1.1 Projekt

Dieses Projekt soll es ermöglichen, eine Java Methode, über Bluetooth, auf einem zweiten Androidgerät auszuführen. Es wird ein Framework erstellt. Dieses hat den Namen: ***BTInvoke***.

1.2 Vorrausetzungen und Anmerkungen

Die Implementierungen wurden in Eclipse Version 4.4.1 mit dem aktuell neusten ADK durchgeführt. Das *Target API Level* war 19.

Um das erstellte Beispielapp korrekt benutzen zu können sind zwei Androidgeräte notwendig, die mindestens API Level 14 und Bluetooth unterstützten. Es wird empfohlen das App von Eclipse aus zu starten. Ansonsten muss ein APK erstellt und auf beiden Geräten installiert werden.

1.3 Aufbau des Dokumentes

In Kapitel 2 wird das Projekt und die Ziele beschrieben. In Kapitel 3 wird danach auf die Implementierung des Frameworks eingegangen. Probleme und Lösungen werden vorgestellt. Kapitel 4 enthält eine Beschreibung und ein Tutorial des erstellten Beispielapps. Im letzten Kapitel 5 wird ein Fazit zum Projekt gegeben und Verbesserungsvorschläge sowie mögliche Erweiterungen genannt.

2 Projektbeschreibung

Dieses Kapitel beschreibt die Ziele des Projektes. Es beantwortet die Frage: Was soll implementiert werden?

Das Hauptziel des Projektes ist die Implementierung eines Frameworks, das es erlaubt Berechnungen, oder sonstige Aktionen mit hoher Rechenleistung, auf ein zweites Android Gerät zu verteilen. Ein Gerät agiert als der Auslöser der Aktion. Auf ihm wird lediglich ein UI angezeigt, mit dem die Aktion gestartet werden kann. Das zweite Gerät wartet auf Anfragen des UI-Gerätes und führt diese aus. Danach sendet das berechnende Gerät eine Antwort zurück zum UI-Gerät. Dieses kann dem Benutzer dann das Ergebnis anzeigen.

Ein Beispielszenario wäre folgendes: Man hat eine Smartwatch, sowie ein reguläres Smartphone. Über das UI auf der Smartwatch wird eine Aktion gestartet. Die dahinter liegenden Berechnungen werden allerdings nicht auf der Smartwatch ausgeführt, sondern auf dem Smartphone, das üblicherweise einen schnelleren Prozessor beinhaltet. Somit ist es möglich schneller das Ergebnis zu erhalten und auch Batterieladung auf der Smartwatch zu sparen.

Das sekundäre Ziel des Projektes ist die Gewinnung von Erfahrung in Androidprogrammierung für den Autor.

3 Implementierung des Projektes

Dieses Kapitel beschreibt, wie das Projekt umgesetzt wurde. Als erstes wirds auf verwendete Fremdframeworks und die zum Testen verwendeten Geräte eingegangen. Im letzten Abschnitt wird die Umsetzung erläutert.

3.1 Verwendete Frameworks

In diesem Projekt wurde das Framework *Android Annotations* [Ric+15] verwendet. Mithilfe von *Java Annotationen*, werden viele Arbeiten, die sich bei der Androidprogrammierung oft wiederholen, automatisch erledigt. Dazu wird Codegenerierung verwendet. Hauptsächlich wurde es für die Annotation `@Background` [Ric15] verwendet. Eine so annotierte Methode wird automatisch auf einem Hintergrund Thread ausgeführt. Somit müssen sich keine Gedanken um `AsyncTask` oder eine eigene Implementierung von Threads gemacht werden.

3.2 Verwendete Geräte

Während der Implementierung wurden zwei Android Smartphones verwendet: Ein *Google Nexus 5* mit Android 4.4.4 und ein *Google Nexus One*, das dank rooting und Custom Rom Android 4.4 installiert hat. Das Nexus One wurde als das UI-Gerät und das Nexus 5 als das berechnende Gerät verwendet. Es wurde jeweils das Beispielapp ausgeführt. Nähere Information zur Beispiel App befinden sich in Kapitel 4.

3.3 Umsetzung

Nach einer Recherche wurde entschieden, Bluetooth für den Austausch von Nachrichten zwischen den Geräten zu verwenden. Eine alternative ist *Wi-Fi Direct* bzw. *Wi-Fi Peer-to-Peer* [Goo15b]. Dies ist allerdings nur auf neueren Geräten unterstützt, sodass es nicht verwendet werden konnte. Auch die lange Reichweite und Bandbreite die es erlaubt, ist nicht notwendig. Die Reichweite und Bandbreite von Bluetooth ist ausreichend für dieses Projekt.

Für die Form der Nachrichten wurde JSON gewählt. Dies wurde entschieden, als in der anfänglichen Recherche *JSON-RPC* [JSO13] gefunden wurde. Das JSON-Format der Nachrichten wurde diesem Nachempfunden. Abbildung 3.1 zeigt das Format einer Anfrage.

Solch eine Anfrage wird dem berechnenden Gerät geschickt. Das Gerät führt die Methode aus und schickt eine Antwort zurück. Abbildung 3.2 zeigt das JSON-Format der Antwort.

```
{
  "id" : 0,
  "method" : "exampleMethod",
  "params" : [42]
}
```

Abbildung 3.1: Anfrage zur Ausführung der Methode `exampleMethod` mit einem Parameter

```
{
  "id" : 0,
  "method" : "exampleMethod",
  "result" : 4711
}
```

Abbildung 3.2: Antwort auf die Anfrage. Das Ergebnis des Methodenaufrufs war 4711

Die folgenden Unterabschnitte gehen auf die zu lösenden Probleme und deren Lösungen ein. Im letzten Unterabschnitt befindet sich ein Tutorial zur Benutzung von BTInvoke.

3.3.1 Die Bluetoothverbindung

Eine Bluetoothverbindung funktioniert nach dem *Client-Server* Prinzip. Das Servergerät akzeptiert Verbindungen und das Clientgerät verbindet sich. Dies wurde in der abstrakten Klasse `BTConnection` und dessen Unterklassen `BTCServerConnection` und `BTClientConnection` umgesetzt. Um die Lösung nicht zu kompliziert zu gestalten, wird davon ausgegangen das die beiden Geräte bereits durch *Pairing* verbunden und einander bekannt sind. Weitere Information zu der Implementierung der Klassen befinden sich in der erstellten Javadoc-Dokumentation. Alle Klassen die sich um die Bluetoothverbindung kümmern sind im Package `btinvoke.bluetooth`. Bei der Implementierung der Bluetoothverbindung hat vor allem das *BluetoothChat* Beispiel von Google geholfen [Goo14a].

Ein Problem, ist die Erhaltung der Verbindung. Falls sich die Verbindung in einem Activity befindet, würde sie zerstört werden, wenn auch die Activity zerstört wird, wie z. B. bei einer Drehung des Gerätes. Um dies zu verhindern und die Verbindung während der gesamten Lebenszeit der Applikation zu erhalten, gibt es einige Möglichkeiten. Darunter sind:

- Aufbau der Verbindung in einer eigenen `Application` Klasse.
- Aufbau der Verbindung in einem UI losen *Fragment*. [Goo15a]
- Aufbau der Verbindung in einem *Service*

Da das Aufrechterhalten ein Hintergrundprozess ist und kein UI benötigt, wurde sich für die Servicevariante entschieden. Dieser kann zusätzlich mit Broadcasts über den Status des Bluetooth Adapters und der Verbindung informiert werden und reagieren. Außerdem ist sein Lebenszyklus nicht an den einer Activity gebunden.

Es wurden zwei Services, für Server- und Clientseite, geschrieben: `BTInvokeServerService`, der die Bluetoothverbindung direkt innerhalb von `onCreate` aufbaut und

`BTInvokeClientService`, der die Verbindung erst aufbaut wenn er per `startService` ein Intent mit der *Action* `ACTION_CONNECT` erhält. Dies ist notwendig, da die `connect` Methode der Bluetooth-Klasse einen Timeout besitzt. Es soll genug zeit bleiben um die Applikation auf dem Servergerät zu starten und dann erst die Verbindung aufzubauen. Dies erlaubt außerdem eine Neuverbindung, falls diese zusammengebrochen ist. Beide Services müssen über `startService` gestartet werden. Die Bindung ist nicht erlaubt, die Kommunikation findet über *Broadcasts* statt. Dies soll zur Entkopplung von Activity und Service beitragen. Ein gebundener Service, und damit die Bluetoothverbindung, würde außerdem nur so lange am Leben erhalten werden, wie er gebunden ist. Weitere Informationen lassen sich im Javadoc finden.

Abbildung 3.3 und Abbildung 3.4 zeigen vereinfachte Sequenzdiagramme des Verbindungsaufbaus auf beiden Seiten.

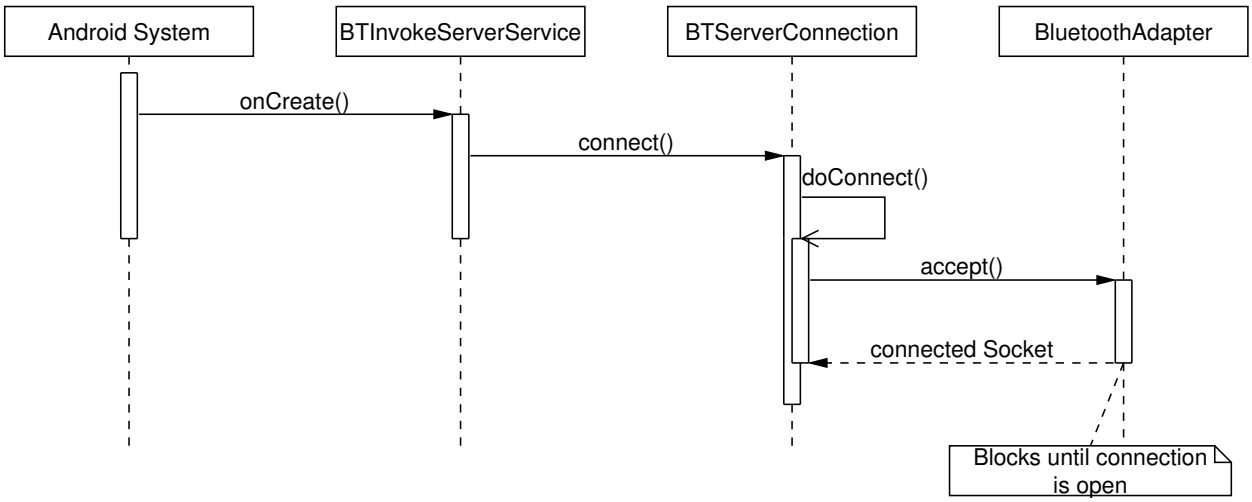

Abbildung 3.3: Verbindungsaufbau Serverseite

3.3.2 Aufruf von Methoden über Bluetooth

Um einen String über Bluetooth zu senden, braucht das jeweilige Objekt Zugriff auf den `InputStream` und `OutputStream` der Bluetoothverbindung. Da das Verbindungsobjekt innerhalb der Services existiert, wurde eine Möglichkeit benötigt den zu sendeten String zum Service zu schicken. Da die Services bereits *Broadcasts* und *BroadcastReciever* verwenden, schien es sinnvoll diese zu verwenden.

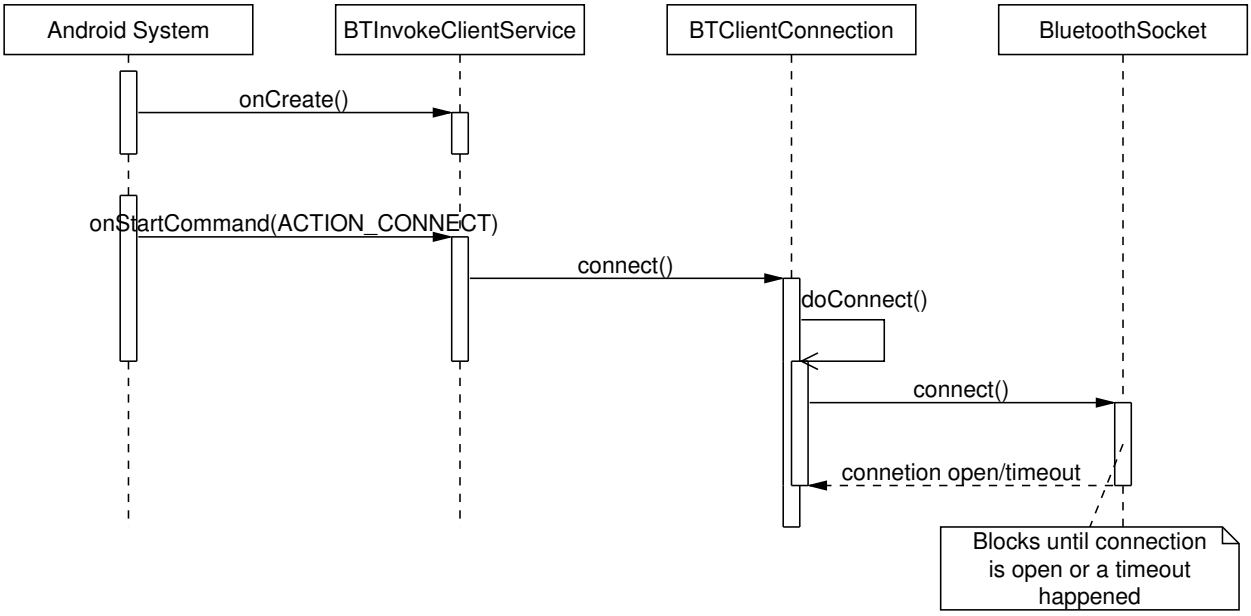

Abbildung 3.4: Verbindungsaufbau Serverseite

Der Aufruf einer Methode über Bluetooth läuft wie folgt ab:

1. Benutzer/Activity auf Server- bzw. UI-Seite ruft die statische Methode `BTInoke.remoteExecute` auf.
2. `BTInoke.remoteExecute` erstellt ein `RemoteInvocationRequest` und sendet ihn als JSON-String über einen Broadcast mit der *Action* `BTInvokeMessages.REMOTE_INVOCATION`.
3. `BTInvokeServerService` bekommt den Broadcast, sendet den JSON-String über Bluetooth und wartet auf eine Antwort.
4. Auf dem Clientgerät empfängt `BTInvokeClientService` den JSON-String, konvertiert ihn zurück zu einem `RemoteInvocationRequest`-Objekt und ruft die Methode mit dem darin enthaltenen Namen und Parametern auf.
5. Nachdem die Methode fertig ist, erstellt `BTInvokeClientService` aus dem Ergebnis ein `RemoteInvocationResult`-Objekt und sendet es als JSON-String zurück zum Servergerät.
6. `BTInvokeServerService` empfängt den String und sendet ihn in einem Broadcast mit der *Action* `BTInvokeMessages.REMOTE_INVOCATION_RESULT`.
7. Die Activity die den Aufruf gestartet hat kann nun diesen Broadcast empfangen und daraus das Ergebnis des Methodenaufrufs holen.

Abbildung 3.5 zeigt den Ablauf nochmal als vereinfachtes Sequenzdiagramm. Weitere Informationen zu den beteiligten Klassen befinden sich im Javadoc.

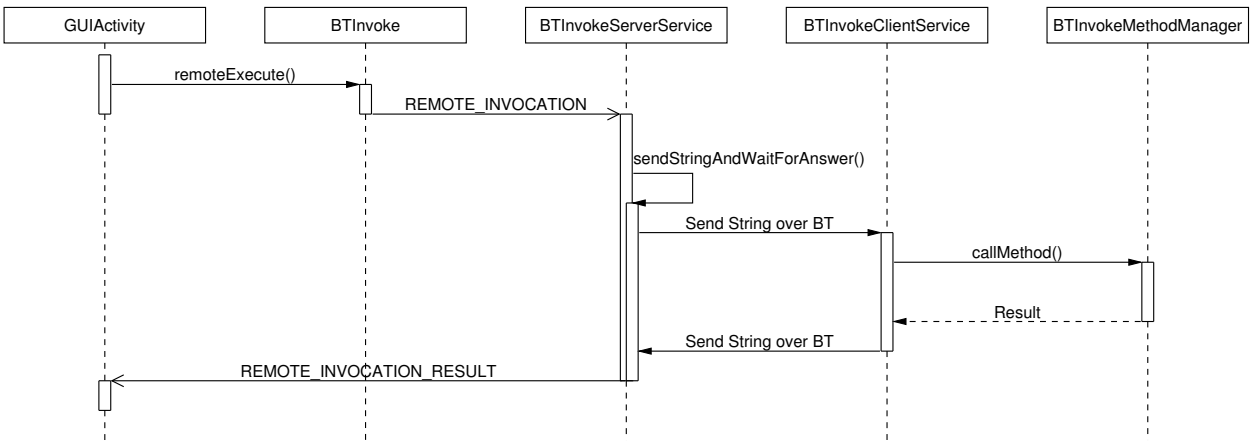

Abbildung 3.5: Aufruf einer Methode über Bluetooth

Bevor eine Methode auf der Clientseite ausgeführt werden kann, muss erst ein Interface, das sie beinhaltet, mit der Methode `registerInterfaceAndImplementation` der Klasse `BTInvokeMethodManager` registriert werden. Hier gibt es allerdings einige Einschränkungen:

- Es können nicht zwei Interfaces registriert werden, die eine Methode mit gleichem Namen besitzen. In so einem Fall würde versucht werden, die erste gefundene auszuführen.
- Eine Methode in einem Interface darf nicht überladen sein.
- Die Methoden können nur primitive Datentypen und String als Parameter oder Rückgabetyp besitzen.

In der aktuellen Version bekommt die Activity den JSON-String mit dem Broadcast geschickt und muss das Ergebnis selbst herausholen. Dies ist unschön, allerdings konnte eine bessere Lösung aus Zeitgründen nicht mehr Implementiert werden.

Es wurde zusätzlich noch eine zweite Variante implementiert, die auf dem vorigen beschriebenen Ablauf aufbaut. Die Klasse `Proxy` aus der Java Standard Bibliothek erlaubt es, zur Laufzeit ein Interface zu implementieren. Alle Aufrufe über die so erstellte Klasse, werden von einem Objekt ausgeführt, das das Interface `InvocationHandler` implementiert [dpu02]. Somit wirkt der Aufruf für einen Benutzer wie ein lokaler Aufruf. Für diesen Zweck wurde die Klasse `BTInvocationHandler` erstellt. Sie lässt sich wie in Abbildung 3.6 dargestellt benutzen:

```
// Implemetiere Interface zur Laufzeit
IExampleInterace e = Proxy.newProxyInstance(getClass().getClassLoader(), new
    Class<?>[]{IExampleInterace.class}, new BTInvocationHandler(context);
// Jetzt ist es moeglich eine Methode des Interfaces aufzurufen als waere sie lokal.
int result = e.exampleMethod(42);
```

Abbildung 3.6: Beispiel der Proxy-Variante

Wichtige Anmerkung: Innerhalb von `BTInvocationHandler` wird einfach der zuvor genannte Ablauf mit `BTInoke.remoteExecute` gestartet. Damit allerdings auch das Ergeb-

nis zurückgegeben werden kann, muss der Aufruf so lange blockieren, bis der Broadcast `BTInvokeMessages.REMOTE_INVOCATION_RESULT` empfangen wurde. Ein BroadcastReciever wird immer im UI-Thread aufgerufen. Wird nun eine Methode des Interfaces auch auf dem UI-Thread aufgerufen, ist dieser blockiert und der BroadcastReciever kann niemals ausgeführt werden. Somit ist die ganze Applikation blockiert.

3.3.3 Tutorial: Benutzung von BTInvoke

Ein Benutzer muss die folgenden Schritte ausführen um BTInvoke zu benutzen:

1. Clientseite: Registrieren eines Interface mit einer implementierenden Klasse in `BTInvokeMethodManager`.
2. Clientseite: Starten des `BTInvokeClientService`.
3. Serverseite: Starten des `BTInvokeServerService`.
4. Clientseite: Starten des Verbindungsaufbaus mit `startService` und dem `ACTION_CONNECT` Intent.
5. Serverseite, falls Proxyvariante: Erstellung einer Proxyimplementierung mit `Proxy.newProxyInstance` und `BTInvocationHandler`.
6. Serverseite: Starten des Methodenaufrufs durch Broadcast oder Proxyvariante.
7. Serverseite: Warten auf Antwort und Ergebnis verwenden.

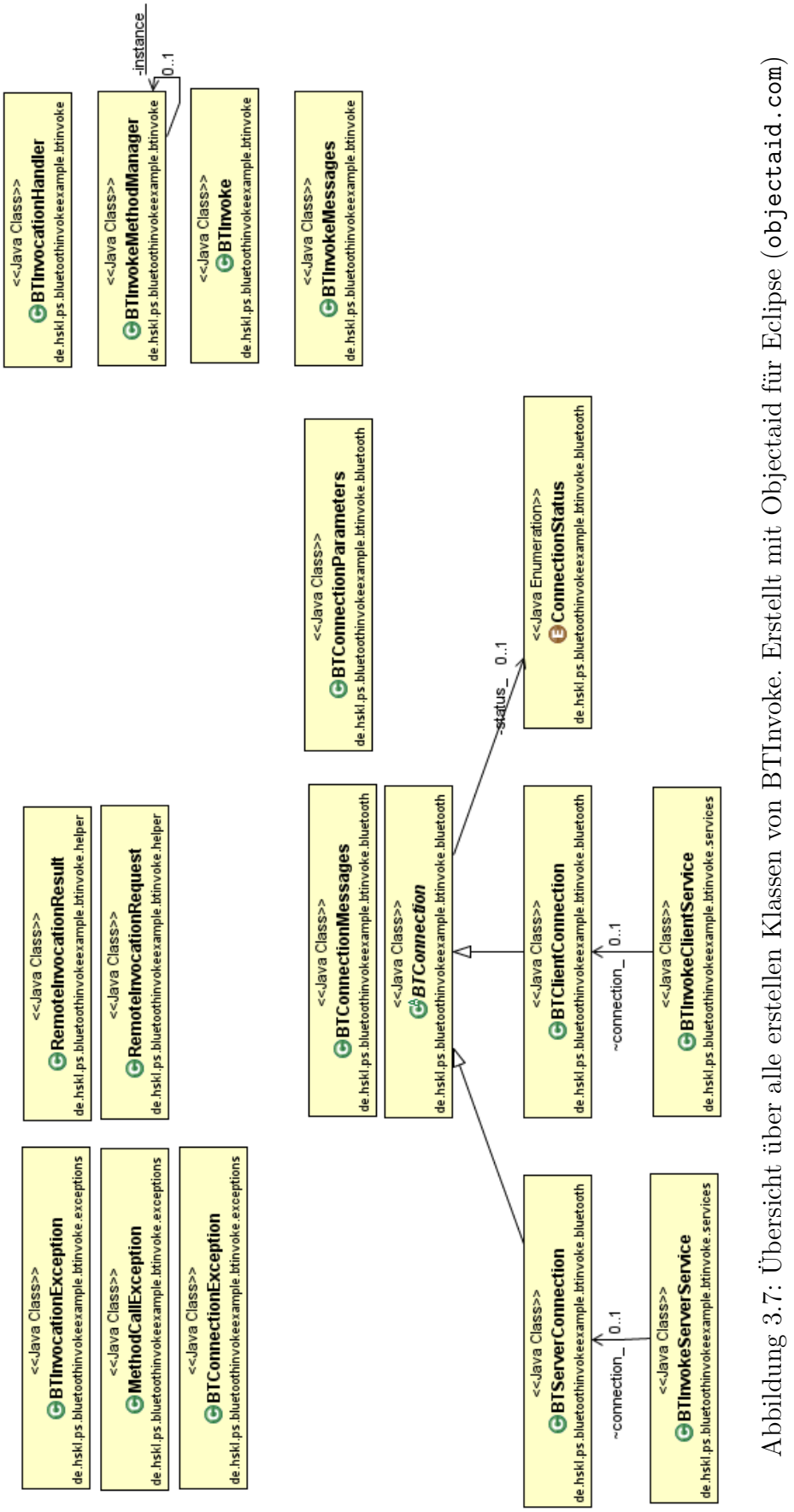

Abbildung 3.7: Übersicht über alle erstellen Klassen von BTInvoke. Erstellt mit Objectaid für Eclipse (`objectaid.com`)

4 Beispielapp

Während der Entwicklung wurde ein Beispielapp geschrieben, mit dem auch die Implementierung getestet wurde. Wenn dieses gestartet wird, sieht man das in Abbildung 4.1 dargestellte UI. Client- und Serverseite wurden der Einfachheit halber in einem App kombiniert. Durch drücken des jeweiligen Buttons kann entschieden werden, welche Seite das Gerät übernehmen soll.

Abbildung 4.1: UI von `ComputeActivity`

Die Activity der Serverseite hat die in Abbildung 4.2 gezeigte Oberfläche. Im oberen Teil des Bildschirmes werden Statusnachrichten angezeigt, die von der Bluetoothverbindung und den Services verschickt werden. Im unteren Teil sind drei Buttons welche jeweils ein Methodenaufruf über Bluetooth verursachen. Die ersten beiden arbeiten direkt mit dem BroadcastReciever

in der Activity während der dritte Knopf die Proxy Variante benutzt. Der Aufruf findet dank *Android Annotations* in einem Hintergrund Thread statt.

Abbildung 4.3 zeigt die Oberfläche auf der Clientseite. Hier zeigt der obere Teil ebenfalls Statusmeldungen. Der *Connect* Button sendet ein Intent mit `ACTION_CONNECT` zum `BTInvokeClientService`. Dies muss geschehen bevor auf der Serverseite die Beispiele, mithilfe der Buttons, ausgeführt werden können.

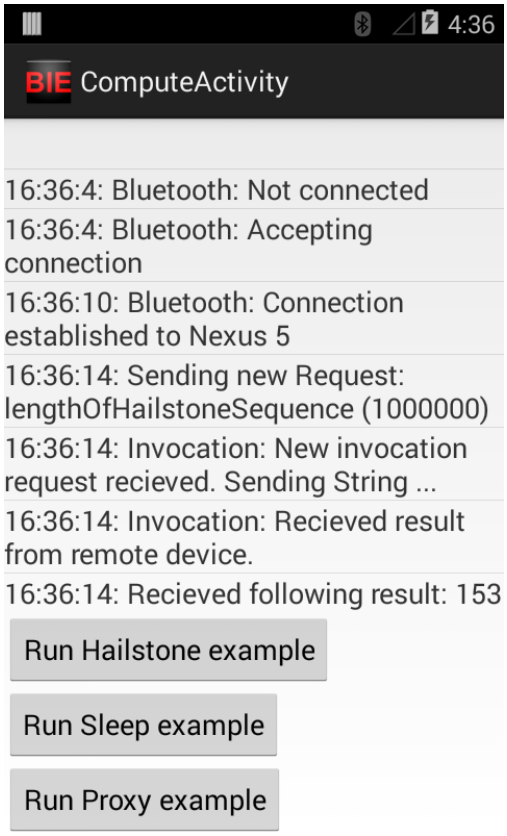

Abbildung 4.2: UI von `GUIActivity`

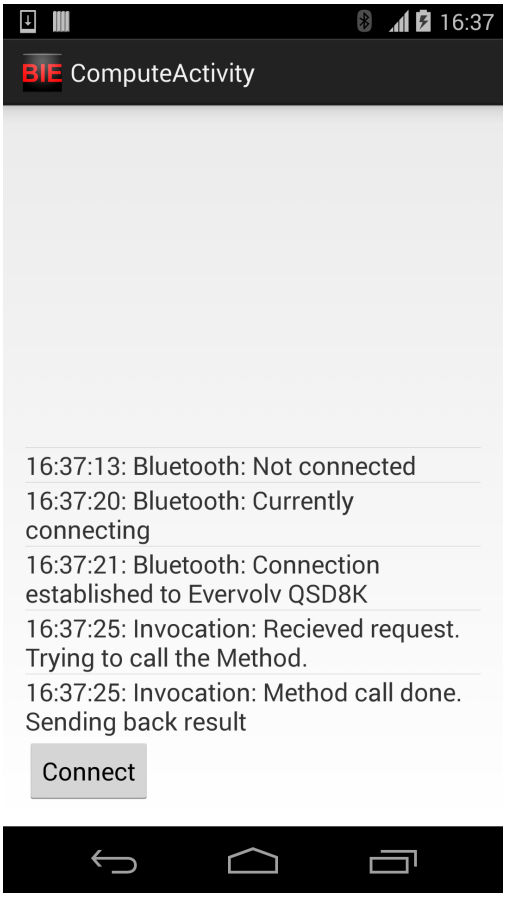

Abbildung 4.3: UI von `StartActivity`

5 Fazit und Ausblick

Dieses Kapitel enthält ein Fazit zum Projekt sowie Verbesserungsmöglichkeiten und mögliche Erweiterungen.

5.1 Fazit

Im Rahmen dieses Projektes wurde ein Framework, oder zumindest ein Proof-Of-Concept entwickelt, das es erlaubt Methoden auf externen Androidgeräten auszuführen und das Ergebnis zurückzubekommen.

Der Autor konnte in diesem Projekt einiges an Erfahrung in der Androidprogrammierung gewinnen. Speziell in den folgenden Bereichen konnte wurde viel Wissen angeeignet:

- Services
- Activity/Service Lebenszyklus
- Broadcasts und BroadcastReciever
- Android Bluetooth API

5.2 Verbesserungsmöglichkeiten und mögliche Erweiterungen

Im Nachhinein wurden noch einige Verbesserungsmöglichkeiten identifiziert. Diese konnten aus Zeitgründen nicht mehr umgesetzt werden.

In der aktuellen Version existieren leider immer noch Situationen, in denen von dem Clientgerät nie eine Antwort geschickt wird. Dies führt auf der Serverseite dazu, das Threads, bis zum Prozessende, nie beendet werden. Hier sollte am besten ein Timeout verwendet werden.

Die Serverseite erlaubt keine Neuverbindung falls diese zusammengebrochen ist. Hier müsste einfach bei einem *Disconnect*-Event die entsprechende Methode wieder aufgerufen werden.

Die Lösung zur Narichtenverteilung mit Broadcasts ist unschön. Es sind viele Konstanten notwendig die anzeigen um welche Art von Nachricht es sich handelt, welche weiteren Informationen in ihr stecken usw.. Auch andere Standardmöglichkeiten von Android, wie die Klassen `Messanger` und `Handler` bieten kein zufriedenstellendes Interface. Hier sollte am besten auf eine Frembibliothek wie *EventBus* [gre15] oder *Otto* [Squ13] zurückgegriffen werden, die es

erlauben eigene Klassen für die Eventarten zu schreiben. Dies ist wesentlich sicherer als das Hantieren mit Strings und Integern.

Der Code ist an vielen stellen noch nicht hundertprozentig sauber. Die Services haben beispielsweise mehrere Verantwortungen und sollten weiter refaktoriert werden.

Auch mögliche Erweiterungen wurden identifiziert:

Aktuell sind nur Methoden aufrufbar, die: nicht überladen sind; nicht doppelt existieren und die nur primitive Typen und String verwenden. Dies ist ein offensichtlicher Punkt der verbesserungswürdig ist. Um Methoden mit gleichem Namen unterscheiden zu können müssen zusätzlich die deklarierten Datentypen betrachtet werden. Diese könnten z. B. als String mitgesendet und dann mit `Class.forName()` wieder hergestellt werden. Hier müssen jedoch Dinge wie Autoboxing und die Unterscheidung zwischen `int` und `long` sowie `double` und `float` besonders betrachtet werden. Für das Übertragen von eigenen Klassen als Parameter könnte *GSON* [Goo14b] verwendet werden.

Interessant wäre auch die Möglichkeit mehrere Clientgeräte zu verbinden. Anfragen müssten somit auf diese verteilt werden.

Quellen

[dpu02] dpunkt.Verlag. *Dynamische Proxies*. Programmierhandbuch und Referenz für die Java-Plattform. 2002. URL: http://www.dpunkt.de/java/Die_Sprache_Java/Objektorientierte_Programmierung_mit_Java/75.html (besucht am 06. 02. 2015) (siehe S. 9).

[Goo14a] Google Inc. *BluetoothChat Example*. Android Developers. 2014. URL: https://developer.android.com/samples/BluetoothChat/index.html (besucht am 09. 01. 2015) (siehe S. 6).

[Goo14b] Google Inc, *google-gson - A Java library to convert JSON to Java objects and vice-versa*. Google Project Hosting. 2014. URL: https://code.google.com/p/google-gson/ (besucht am 15. 02. 2015) (siehe S. 16).

[Goo15a] Google Inc. *Handling Runtime Changes*. Android Developers. 2015. URL: https://developer.android.com/guide/topics/resources/runtime-changes.html#RetainingAnObject (besucht am 11. 02. 2015) (siehe S. 6).

[Goo15b] Google Inc. *Wi-Fi Peer-to-Peer*. Android Developers. 2015. URL: https://developer.android.com/guide/topics/connectivity/wifip2p.html (besucht am 14. 02. 2015) (siehe S. 5).

[gre15] greenrobot. *greenrobot/EventBus*. Github. 2015. URL: https://github.com/greenrobot/EventBus (besucht am 15. 02. 2015) (siehe S. 15).

[JSO13] JSON-RPC Working Group. *JSON-RPC 2.0 Specification*. 4. Jan. 2013. URL: http://www.jsonrpc.org/specification (besucht am 14. 02. 2015) (siehe S. 5).

[Ric+15] Pierre-Yves Ricau, Mathieu Boniface, Alexandre Thomas, Joan Zapata, Romain Sertelon und Damien Villeneuve. *AndroidAnnotations*. 2015. URL: http://androidannotations.org/ (besucht am 14. 02. 2015) (siehe S. 5).

[Ric15] Pierre-Yves Ricau. *How to stop worrying about threads*. excilys/androidannotations Wiki. 2015. URL: https://github.com/excilys/androidannotations (besucht am 14. 02. 2015) (siehe S. 5).

[Squ13] Square, Inc. *Otto*. 2013. URL: https://square.github.io/otto/ (besucht am 15. 02. 2015) (siehe S. 15).